

X/O/P/E/N

PORTABILITY GUIDE

DATA MANAGEMENT

X/O/P/E/N


PORTABILITY GUIDE
DATA MANAGEMENT

10

X/O/P/E/N/

PORTABILITY GUIDE

DATA MANAGEMENT



© 1987, The X/OPEN Group Members

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior permission of the copyright owners.

X/OPEN PORTABILITY GUIDE

Set of 5 Volumes

ISBN: 0-444-70179-6

Volume 1	XVS Commands and Utilities	ISBN: 0-444-70174-5
Volume 2	XVS System Calls and Libraries	ISBN: 0-444-70175-3
Volume 3	XVS Supplementary Definitions	ISBN: 0-444-70176-1
Volume 4	Programming Languages	ISBN: 0-444-70177-X
Volume 5	Data Management	ISBN: 0-444-70178-8

Published by:

ELSEVIER SCIENCE PUBLISHERS B.V.
P.O Box 1991
1000 BZ Amsterdam
The Netherlands

Sole distributors for the U.S.A. and Canada:

ELSEVIER SCIENCE PUBLISHING COMPANY, INC.
52 Vanderbilt Avenue
New York, N.Y. 10017
U.S.A.

Any comments relating to the material contained in the X/OPEN Portability Guide may be submitted to the X/OPEN Group by letter via the Publisher or directly by Electronic Mail to:

xopen@inset.co.uk

PRINTED IN THE NETHERLANDS



Contents

PREFACE


THE COMMON APPLICATIONS ENVIRONMENT

INDEXED SEQUENTIAL ACCESS METHOD (ISAM)

1. Introduction
2. Isam Overview
3. Data Types
4. Indexing
5. Locking
6. C Program Examples
7. Exception Handling
8. The isam.h Header File
9. Call Specifications

RELATIONAL DATABASE LANGUAGE (SQL)

1. Introduction
2. Concepts
3. Common Elements
4. Embedded Aspects
5. Executable Sql Statements
6. Implementation-Specific Issues
 - A. Syntax Summary
 - B. ANS X3-135 Database Language (SQL)
 - C. Future Directions



Trademarks

UNIX[™] is a registered trademark of AT&T in the USA and other countries.

C-ISAM[™] is a trademark of Informix Corporation.

LEVEL II COBOL[™] is a trademark of Micro Focus Limited.

XENIX[™] is a trademark of Microsoft Inc.

IBM[™] is a trademark of International Business Machines Corp.

X/OPEN[™] is a licensed trademark of the X/OPEN Group Members.

POSIX[™] is a trademark of the Institute of Electrical and Electronic Engineers Inc.



Preface

X/OPEN represents a major breakthrough in the world of standards for the information technology industry. Ten of the world's major information system suppliers have come together to encourage applications portability resulting in tangible benefits for computer users, independent software houses and for the suppliers themselves.

The Group's principal aim is to increase the volume of applications available and to maximise the return on investments in software development made by Users and Independent Software Vendors.

This is achieved by ensuring portability of application programs at the source code level. Through this portability, users can mix and match computer systems and applications software from many suppliers, and thus investment in applications software is protected into the future.

In order to provide such portability, the Group defines a **Common Applications Environment** built on the interfaces to the **UNIX** operating system, as defined in the AT&T System V Interface Definition, and covering other aspects required of a comprehensive applications interface.

The X/OPEN Portability Guide contains an evolving portfolio of practical standards for application portability. All of the members of X/OPEN guarantee to support the standards defined, leading to:

- Growing portability
- No dependence on a single source - freedom of choice
- Increased application software selection
- More security in software investments
- International support for the *Common Applications Environment*

X/OPEN is not a standards-setting organisation; it is a joint initiative by members of the business community to integrate evolving standards into a common, beneficial and continuing strategy.

• At the time of publication, the membership of the X/OPEN Group was BULL, DEC, ERICSSON, HEWLETT-PACKARD, ICL, NIXDORF, OLIVETTI, PHILIPS, SIEMENS and UNISYS

Issue 2 of the X/OPEN Portability Guide (published in January 1987) comprises five volumes defining the interfaces currently identified as components of the Common Applications Environment.

Volume 1	System V Specification: Commands and Utilities
Volume 2	System V Specification: System Calls and Libraries
Volume 3	System V Specification: Supplementary Definitions XVS Internationalisation XVS Terminal Interfaces XVS Inter-Process Communication XVS Source Code Transfer
Volume 4	Programming Languages C Language COBOL Language
Volume 5	Data Management Indexed Sequential Access Method (ISAM) Relational Database Language (SQL)

In addition, each volume includes an introduction giving the philosophy of the Common Applications Environment and an overview of its components.

This guide is aimed at both the decision makers and the implementation teams of:

- Independent Software Vendors
- Software Houses
- Users
- Equipment Manufacturers

The Guide is designed to sit permanently on the desk, serving as a common reference point for anyone directly concerned with the practical side of software development, namely systems designers, programmers and consultants.

The various parts of the Portability Guide are closely interrelated. Any reference from one part of the definition to another part uses the title of the other part as a reference (e.g., "XVS TERMINAL INTERFACES").

Acknowledgements

X/OPEN gratefully acknowledges:

- **AT&T** for permission to reproduce portions of its copyrighted System V Interface Definition (SVID) and material from the UNIX System V Release 2.0 documentation.
- The **/usr/group** Standards Committee, whose Standard contributed to the Group's work.
- **Informix Corporation.** of Menlo Park, California (Telex no. 361834) for permission to use material from the specification of their C-ISAM product and for provision of that material in machine readable form.
- **Micro Focus Ltd.** of Newbury, Berkshire for permission to use material from the specification of their LEVEL II COBOL compiler.
- The assistance given by the following companies in the preparation of the Database Language (SQL) definition:

Informix Corporation
Oracle Corporation
Queensland Information Technology
Relational Technology Inc.
Unify Corporation

Referenced Documents

The following documents are referenced in this guide:

- System V Interface Definition (Spring 1985 - Issue 1)
- System V Interface Definition (Spring 1986 - Issue 2)
- UNIX System V Release 2.0 Programmer's Reference Manual (April 1984 - Issue 2)
- UNIX System V - Release 2.0 Programming Guide (April 1984 - Issue 2)
- ANS Draft Proposal for C Language (October 1986 - ANS X3J11/86-151)
- 1984 /usr/group Standard
- IEEE P1003.1 Trial Use Standard (April 1986)
- Informix Corporation C-ISAM Reference Manual (Version 2.10 - January 1985)
- MicroFocus Level II COBOL Language Reference Manual (Version 2.5 and 2.6, Issue 7 - April 1984)
- Standard for COBOL (ANS X3.23-1974)
- Standard for COBOL (ANS X3.23-1985)
- Standard for FORTRAN (ANS X3.9-1978)
- Standard for Database Language (SQL) (ANS X3.135-1986)
- Standard for PASCAL (ISO 7185-1983)

X/O/P/E/N/

PORTABILITY GUIDE

THE COMMON APPLICATIONS
ENVIRONMENT

Contents

Chapter	1	THE COMMON APPLICATIONS ENVIRONMENT
Chapter	2	SYSTEM V
	2.1	INTRODUCTION
	2.2	THE EVOLVING STANDARD
	2.2.1	Origins
	2.2.2	The IEEE "POSIX" Standard
	2.2.3	The AT&T System V Interface Definition
	2.3	THE X/OPEN SYSTEM V SPECIFICATION
	2.3.1	System Calls and Libraries
	2.3.2	Inter-process Communication
	2.3.3	Commands and Utilities
Chapter	3	INTERNATIONALISATION
	3.1	INTRODUCTION
	3.2	The X/OPEN NATIVE LANGUAGE SYSTEM
Chapter	4	C LANGUAGE
	4.1	INTRODUCTION
	4.2	C LANGUAGE PORTABILITY GUIDELINES
	4.3	THE ANS X3J11 DRAFT STANDARD
	4.4	THE C PROGRAM PORTABILITY CHECKER (<i>lint</i>)
Chapter	5	OTHER PROGRAMMING LANGUAGES
	5.1	INTRODUCTION
	5.2	COBOL
	5.3	FORTRAN
	5.4	PASCAL
Chapter	6	DATA MANAGEMENT
	6.1	INTRODUCTION
	6.2	INDEXED SEQUENTIAL ACCESS METHOD (ISAM)
	6.3	RELATIONAL DATABASE LANGUAGE (SQL)

Chapter	7	SOURCE CODE TRANSFER BETWEEN MACHINES
	7.1	INTRODUCTION
	7.2	FLOPPY DISC STANDARD
	7.3	MAGNETIC TAPE
	7.4	UTILITIES
Chapter	8	NETWORKING AND COMMUNICATIONS
	8.1	NETWORKING AND COMMUNICATION
	8.2	OPEN SYSTEMS INTERCONNECTION
	8.3	GENERALISED INTER-PROCESS COMMUNICATION, IPC
	8.4	DISTRIBUTED FILE SYSTEM
	8.5	DISTRIBUTED TRANSACTION PROCESSING

The Common Applications Environment

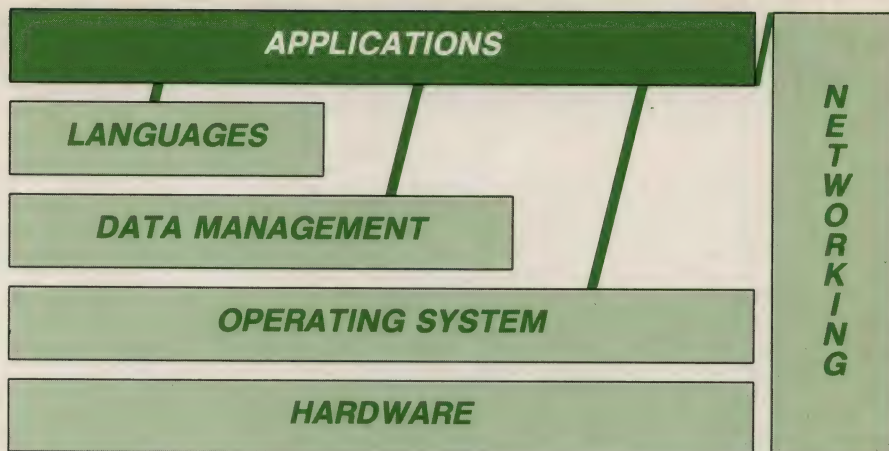
The formation of the X/OPEN Group represents a major initiative by an international group of suppliers of computer systems to create a free and open market, offering Independent Software Vendors (ISVs) as wide a market as possible for their products and giving users an increased return on investment in application software.

The current dominance of proprietary machine environments is restricting the growth of the computer industry. Users tend to get locked into a particular proprietary system by the investment they have made in the applications. Independent Software Vendors are discouraged from writing applications for a particular environment because of the limited markets caused by this fragmentation. This means that there is very little generally available software for each type of system, thus increasing the size of investment needed by each user. All this in turn limits the sales potential of machines from the computer suppliers.

The objective shared by the members of the X/OPEN Group is to establish a Common Applications Environment to the mutual advantage of users, Independent Software Vendors and computer suppliers. Applications written to operate in this environment will be portable at the source code level to a wide range of machines, thereby releasing the user from dependence on a single supplier, reducing the necessary investment in applications, considerably increasing the market for independent software and opening up the market for systems suppliers.

The existence of these "Open Systems" allows users to mix and match systems from different suppliers, and to move applications between machines to meet changing requirements as business grows, thereby giving protection of investment in applications software into the future.

The great increase in the potential market encourages the Independent Software Vendors to produce a wealth of general applications packages, and the availability of this further reduces the investment needed by the users. The whole situation is thus mutually reinforcing.



The foundations of the Common Applications Environment are the interfaces of the UNIX System V operating system, as defined in the AT&T "System V Interface Definition", and the C language.

To define a complete environment for portable applications, it is also necessary to satisfy the requirements for data management, integration of applications, data communications, distributed systems, the use of high level languages and the many other aspects involved in providing a comprehensive applications interface. The X/OPEN Group intends, therefore, to publish progressively definitions covering these areas.

The systems of the X/OPEN Group members that support interfaces derived from UNIX operating systems will do this according to the X/OPEN definitions and will support the full Common Applications Environment.

A specific Common Applications Environment feature may not, however, be present if it is not relevant in the market area in which a particular system is sold. For example, a system sold only in a scientific context might not support COBOL. Conversely, a particular system may support features over and above those of the Common Applications Environment, some of which may partially overlap. An example of this could be that an alternative dialect of COBOL is supported in addition to that of the Common Applications Environment.

The X/OPEN Group is primarily concerned with standards selection and adoption. The general policy is to use International Standards, where they exist, and to adopt "de facto" standards in other cases.

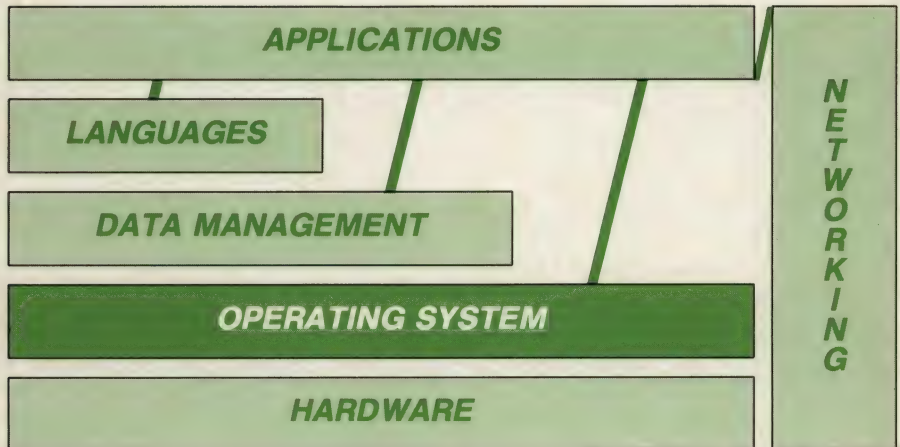
Where International Standards do not exist, it is X/OPEN policy to work closely with standardisation bodies to encourage their emergence.

The Common Applications Environment

It is important that the defined elements of the Common Applications Environment be readily achievable on member systems, and have wide acceptance. For this reason, the definitions, in general, fall within the capabilities of at least one currently available popular product.

In this guide, certain aspects of the Common Applications Environment are defined with reference to the interfaces offered by specific products. This does not mean that member systems will necessarily contain these products, but that the defined interfaces will be supported. Indeed the method of support for an interface on a particular system may change with time.

X/OPEN System V Specification (XVS)



2.1 INTRODUCTION

The X/OPEN System V specification (XVS) defines the applications interfaces provided by the underlying operating system and forms the foundation of the Common Applications Environment.

The XVS is derived from a series of standards activities. The evolving standard is briefly addressed, and the relationship between the X/OPEN System V Specification and the System V Interface Definition and other standards is explained.

2.2 THE EVOLVING STANDARD

2.2.1 Origins

The UNIX operating system was developed by Ritchie and Thompson at Bell Laboratories in the early 1970s. The current AT&T System V version may be traced back directly to that first system.

For many years, it remained basically an academic product. More recently, computer suppliers have adopted the UNIX system as a multi-tasking, multi-user and portable operating environment. They have based their systems on one of several releases, variants or look-alikes. Of these, the most widely used were Version 7, System III, the Berkeley system and XENIX.

Although these systems had much in common, the degree of compatibility at the application interface level was insufficient to permit the development of totally portable applications.

2.2.2 The IEEE "POSIX" Standard

/usr/group, a group of users of UNIX derivatives in the USA, established a committee with the objective of proposing a set of standards for application level interfaces. After publishing its standard, together with a reviewer's guide, the group decided to seek IEEE status for the standard. In late 1984, the /usr/group standards committee closed its activities in its own name and its members were encouraged to become involved in the IEEE group, known as P1003.

The P1003 group published a "trial-use" standard in early 1986, which has the status of a "Draft American National Standard". This "Portable Operating System for Computer Environments" (POSIX) is expected to be revised and submitted for approval in 1987.

The IEEE P1003 group is working to extend the POSIX standard. It is expected that the next area to be standardised will be the subset of commands which offer an interface to applications.

2.2.3 The AT&T System V Interface Definition

The "System V Interface Definition" (SVID), first published by AT&T in the spring of 1985, represented a major standards initiative. AT&T were prominent in the activities of /usr/group and the influence of /usr/group can clearly be seen in the SVID. The stated purpose of the SVID is to define common interfaces for all System V implementations.

The definition groups interfaces into a mandatory *base* plus a series of *extensions*. The *base* interfaces must be present in any implementations of System V. If any interface from an *extension* is supported, it must adhere to the definition.

Issue 1 of the SVID comprised a single volume defining operating system interfaces (known as *system calls* and *library routines*) available to applications as directly called external functions and defined in terms of invocation from C-language programs.

Issue 2 of the SVID was published in early 1986 and comprised two volumes. The first volume contained the same material as issue 1, with some restructuring to improve ease of use and some changes to correct errors. It comprised the *Base System Definition* plus a single extension referred to as the *Kernel Extension*.

The second volume primarily defined commands and utilities, normally invoked through a command interpreter. It comprised further extensions referred to as the *Base Utilities Extension*, *Advanced Utilities Extension*, *Administered Systems Extension*, *Software Development Extension*, and *Terminal Interface Extension*. The latter two include library routines in addition to utilities.

2.3 THE X/OPEN SYSTEM V SPECIFICATION

The X/OPEN System V Specification (XVS) is based upon the AT&T System V Interface Definition, but also taking into account the trial use standard published by IEEE and the capabilities of the existing AT&T System V product.

The XVS is organised into a number of self-contained sections:

"XVS SYSTEM CALLS AND LIBRARIES" defines the Operating System Interfaces and broadly corresponds to Volume 1 of the SVID.

"XVS COMMANDS AND UTILITIES" defines commands and utilities and broadly corresponds to Volume 2 of the SVID. The purpose of the X/OPEN Portability Guide is to facilitate the portability of applications. As such, system administration is outside of its scope and the routines included in the AT&T *Administered System Extension* are not defined.

"XVS TERMINAL INTERFACES" defines a set of portable interfaces to locally connected asynchronous terminals and broadly corresponds to the AT&T Terminal Interface Extension in Volume 2 of the SVID.

"XVS INTER-PROCESS COMMUNICATION" defines interfaces to shared memory, semaphores and message passing, included as an interim mechanism to satisfy the immediate requirements of Inter-Process Communication facilities.

2.3.1 System Calls and Libraries

"XVS SYSTEMS CALLS AND LIBRARIES" contains a full definition of interfaces to system calls and library routines and broadly corresponds to Volume 1 of the SVID.

The X/OPEN Group has extended the SVID in a number of areas:

- Certain changes have been included, which the SVID denotes as future directions.
- The use of symbolic names to replace numeric constants, introduced by AT&T in their SVID, has been extended.
- Clarification of existing wording has been introduced in a limited number of places to "tighten" the specification.
- The opportunity has been taken to correct clerical errors in the SVID.
- Definitions have been included of a number of further UNIX System V Release 2.0 functions which are in widespread use by application developers.

The relationship between the XVS and the SVID is clearly stated. The whole of the SVID *base* definition is included as mandatory with the exception of the maths group, which is not mandatory for systems sold into markets where it is not relevant. *Termio* was not mandatory in issue 1 of the XVS because of some difficulties in implementation. These have now been resolved, and the routine is now mandatory, except in systems which do not support locally connected asynchronous lines.

The XVS incorporates all the interfaces within the SVID *kernel extension set* although a number are defined as optional.

In the XVS, interfaces defined as *optional* will be available on most but not necessarily all X/OPEN systems; use of them could restrict portability. Any *optional* interface supported on an X/OPEN system will conform to the X/OPEN specification.

The XVS defines interfaces in terms of their interface syntax and run-time behaviour, without constraining the method of their implementation. The names "system calls" and "subroutines" are retained purely for compatibility with other documentation.

2.3.2 Inter-process Communication

The kernel extension interfaces relating to shared memory, semaphores and message passing are included in "XVS INTER-PROCESS COMMUNICATION" as a short-term mechanism to satisfy the immediate requirements for Inter-Process Communication facilities. However, they are machine specific and cannot be supported on all hardware architectures. The Group believes that a more generalised approach to the whole subject of Inter-Process Communication is required.

2.3.3 Commands and Utilities

"XVS COMMANDS AND UTILITIES" contains a full definition of interfaces to commands and utilities and broadly corresponds to Volume 2 of the SVID. The interfaces are split functionally into those which are intended to provide an applications interface (referred to as *Standard Utilities*) and those which are only intended to be used by development programmers or during the porting of applications to an X/OPEN system (referred to as *Development Utilities*). The Standard Utilities will be present in all X/OPEN systems, as they are needed to provide a run-time interface to applications. The Development Utilities may only be present in development systems; their respective descriptions are clearly annotated to indicate this. This same distinction is also present in the SVID. The X/OPEN development utilities correspond exactly to the SVID *Software Development Extension*.

The definition of standards for commands and utilities is an evolving process and X/OPEN intends to participate fully.

The current definition is a valuable first step, but additional work will be needed to evolve towards a complete standard. For example:

- The number of options defined for many of the commands is excessive and includes functionality which is rarely used or is implementation specific.
- There are too many different ways of achieving the same results.
- Many of the current descriptions were written to record the observed behaviour of already existing utilities and the level of precision is inadequate for use as a definitive standard.

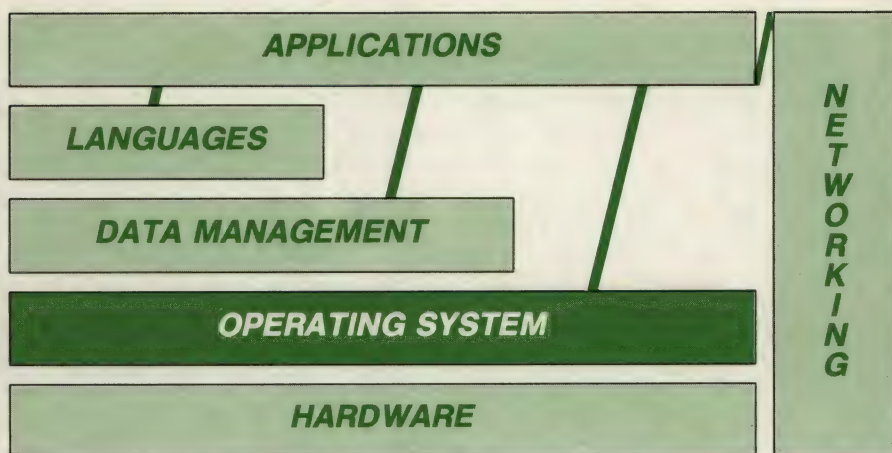
Rectification of this is an enormous task and the current X/OPEN definition is of necessity based on the available documentation, but incorporates extensive annotation to highlight potential portability problems resulting from the points listed above. To achieve maximum portability, application developers should avoid the use of the functionality so annotated.

X/OPEN is working on a substantially improved definition of commands, with the number of options reduced to those in common use and with a higher level of specification. In addition to the current X/OPEN specification, "XVS COMMANDS AND UTILITIES" contains a proposed template for improved specifications together with a number of examples.

This improvement process will be carried out in close consultation with the various user organisations and standards bodies, such as IEEE and ISO, to ensure that the result is a single standard definition of operating system commands and utilities.

Readers of the X/OPEN Portability Guide are invited to participate directly in the consultative process to ensure that the evolving standard matches the requirements of existing and future applications.

Internationalisation



3.1 INTRODUCTION

X/OPEN members market systems in many countries. Our customers and users speak many different languages and conform to different cultural conventions and business practices. It is important therefore that X/OPEN systems are capable of supporting a range of language and cultural environments. In many cases a strong requirement also exists to cope with these variations on the same system. An example is within the administration of the European Economic Community.

To date UNIX operating systems and most systems derived from them have been based on the ASCII 7-bit coded character set and on American English. There are no facilities for dealing with other coded character sets, nor for supporting different languages and cultural conventions.

The requirement for effective mixed language working brings with it the need for coded character sets larger than can be accommodated by 7-bit characters, as does the requirement to support the more complex languages. At the same time there is a trade-off between the ability to handle larger coded character sets, and the amount of storage required to hold the data. For most European requirements an 8-bit system provides the correct balance. For the major Eastern languages (such as Chinese and Japanese) a 16-bit system is necessary, even to support a single language.

To satisfy these requirements enhancements must be made to the system to provide full data transparency to applications, allowing flexibility in the choice of coded character set(s) employed. Additionally, the system must allow program messages (both input and output) to be handled in the native language of each user, as well as providing cultural dependent data items (such as date formats and currency symbols).

3.2 THE X/OPEN NATIVE LANGUAGE SYSTEM

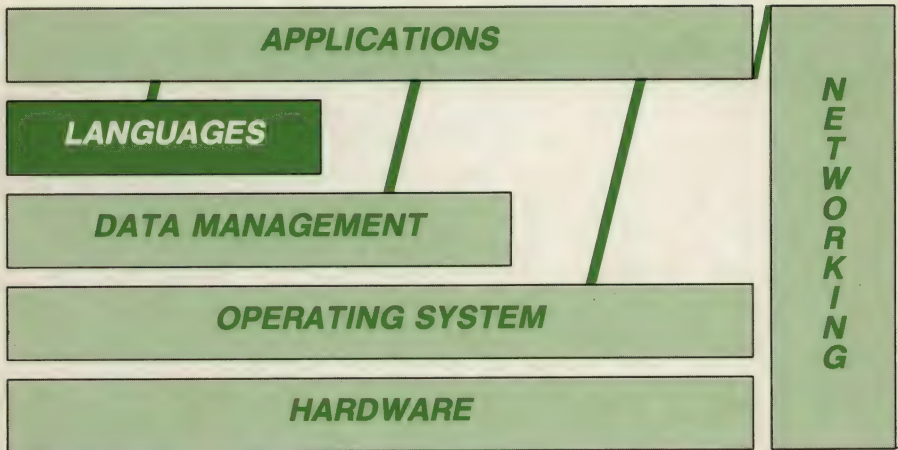
The X/OPEN Native Language System (NLS) is a set of interfaces designed to facilitate the development of applications that can operate in many different language and cultural environments. The interfaces have been derived from those of the Native Language Support system developed by the Hewlett-Packard Company of Palo Alto, California. They have been further enhanced by X/OPEN and have been modified in strategic areas to more closely relate to the Internationalisation proposals of the Draft Proposed American National Standard for the C Programming Language.

The first issue of the specification, defined in "XVS INTERNATIONALISATION", concentrates on facilities for the development of internationalised applications (rather than on internationalising the operating system itself), and on the 8-bit coded character set situations.

The following groups of facilities are defined:

- A message catalogue system which allows program messages to be held apart from the program logic, translated into different native languages, and the appropriate version retrieved by the program at run time.
- An announcement mechanism whereby native language, local custom (territory) and codeset requirements appropriate to each user can be identified to applications at run time.
- Enhanced interface definitions of standard C library functions, which provide language dependent character type classification, upper to lower case and lower to upper case character conversions, date and time messages, floating point to string conversions, and text collation.
- Library functions which allow programs to determine cultural and language specific data dynamically (e.g. the format of date and time strings, weekday and month names, currency symbols, etc.).
- A set of standard commands and library functions which will operate correctly with 8-bit characters.

C Language



4.1 INTRODUCTION

This chapter addresses the C language and guidelines for portability when writing C code.

Currently the American National C Language Standards committee, X3J11, is working towards a standard for the C programming language. The X/OPEN Group is represented on that committee by member companies and intends to adopt the standard, once it has been established as a practical reality.

Meanwhile, the X/OPEN definition included in "C LANGUAGE" is based upon that given in Chapter 2 of the "System V Programming Guide", Release 2.0, published by AT&T.

4.2 C LANGUAGE PORTABILITY GUIDELINES

Whilst the C language provides the basis for applications portability, it is easy to write statements, using valid C constructs, that are machine specific. Care has to be taken when writing programs that are intended to be portable across a range of systems. "C LANGUAGE" includes advice towards ensuring portability.

4.3 THE ANS X3J11 DRAFT STANDARD

The ANS X3J11 standard has been published in a draft form but may still change before it is approved. However, it is already clear that the standard will impose certain restrictions such that programs written to the current C language definition may not work correctly, if the source is later passed through a compiler that supports the ANS standard.

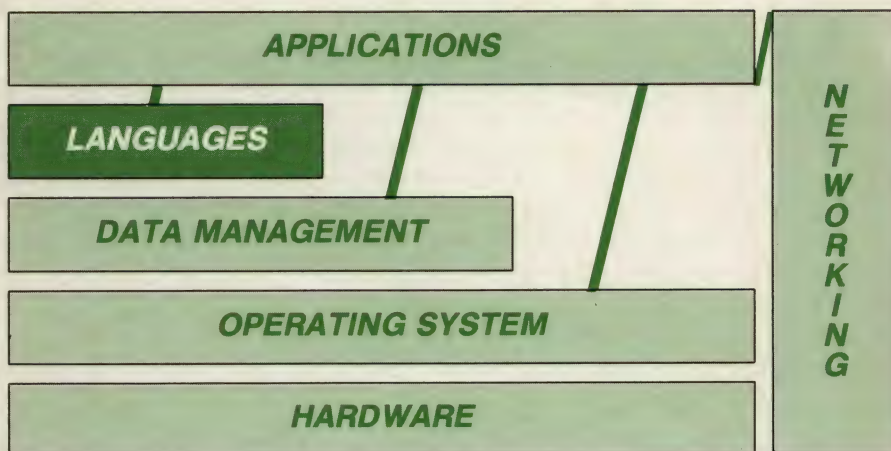
To address this, "C LANGUAGE" includes advice on writing programs to avoid these problems.

4.4 THE C PROGRAM PORTABILITY CHECKER (*lint*)

The *lint* program checks C source programs for violation of most of the portability rules. It also gives a more stringent enforcement of the type rules of C than is provided by most C compilers. A further option detects a number of wasteful or error-prone constructions which nevertheless are syntactically correct.

Use of the *lint* program is recommended: it is described in "C LANGUAGE".

Other Programming Languages



5.1 INTRODUCTION

This chapter addresses programming languages other than C on X/OPEN systems. It covers the inclusion of the principal high level languages in the Common Applications Environment.

To date, The X/OPEN Group has established definitions for COBOL and FORTRAN and PASCAL.

5.2 COBOL

The X/OPEN COBOL definition identifies a common set of language facilities that will be supported by COBOL compilers on all member systems. Applications written to this definition will be portable to any X/OPEN system.

The ISO Working Group and ANS COBOL committee have been working for some years towards a revised standard for COBOL to reflect more accurately the capabilities of modern COBOL compiling systems. The latest international standard, "X3.23 - 1985" was approved during 1985. At the time of publication of Issue 2 of the Portability Guide, there are few compilers in compliance with the revised standard and hence the X/OPEN group feels that major changes in the X/OPEN definition to reflect the new standard would be premature. It is likely, however, that any future edition of the X/OPEN COBOL language definition will relate to "COBOL 1985" and the new standard has been used as a reference when eliminating obsolete elements from the COBOL definition.

The most widely followed standard for COBOL is still that defined in the earlier 1974 Standard, "ANS X3.23-1974", to which most current COBOL compilers substantially conform.

The 1974 standard is incomplete in the area of facilities for interaction with the on-line user. To overcome this deficiency, most COBOL compilers provide extensions to the *ACCEPT* and *DISPLAY* verbs, but they do this in incompatible ways. Since the majority of applications now include interactive operation, it is necessary for a standard form of *ACCEPT* and *DISPLAY* to be defined in the X/OPEN Common Applications Environment.

In order to have an X/OPEN definition that is achievable on member systems within a short timescale, and one that would have immediate widespread acceptance, it has been based on the definition of COBOL embodied in a popular product: Micro Focus LEVEL II COBOL, which itself conforms to the "ANS X3.23-1974".

The Micro Focus LEVEL II COBOL language specification includes a number of other extensions beyond the 1974 standard, in addition to those to *ACCEPT* and *DISPLAY*. None of these are currently included in the X/OPEN definition. The X/OPEN definition also applies restrictions to the ANS-based parts of the LEVEL II definition.

Whilst the X/OPEN COBOL definition is based on the specification of a particular product, the means of implementation across the systems of the X/OPEN members may vary. Any particular system may support extensions beyond the facilities identified, but their use is likely to impede portability.

The X/OPEN COBOL definition is given in detail in "COBOL LANGUAGE" and its relationship to the 1974 standard is clearly shown.

The definition is given in terms of the command syntax derived from the LEVEL II COBOL Reference Manual. The semantics of the *ACCEPT* and *DISPLAY* verbs are defined in "COBOL LANGUAGE". The semantics of all other elements of the language are defined by the "X3.23-1974" standard.

5.3 FORTRAN

The X/OPEN definition for FORTRAN is the formal definition given in the American National Standards document "FORTRAN 77, ANS X3.9 - 1978". This has had wide-scale acceptance throughout the world and there are many certified compilers available.

The majority of FORTRAN compilers, while adhering to the basic FORTRAN 77 standard, also offer extensions beyond that standard. There is little compatibility in these extensions between compilers and they do not form part of the X/OPEN definition. Developers are warned that use of these extensions will affect the portability of FORTRAN programs.

5.4 PASCAL

The current X/OPEN definition for PASCAL is the formal definition given in the International Organisation for Standardisation document "Programming Languages - PASCAL " ISO 7185-1983 (level 1).

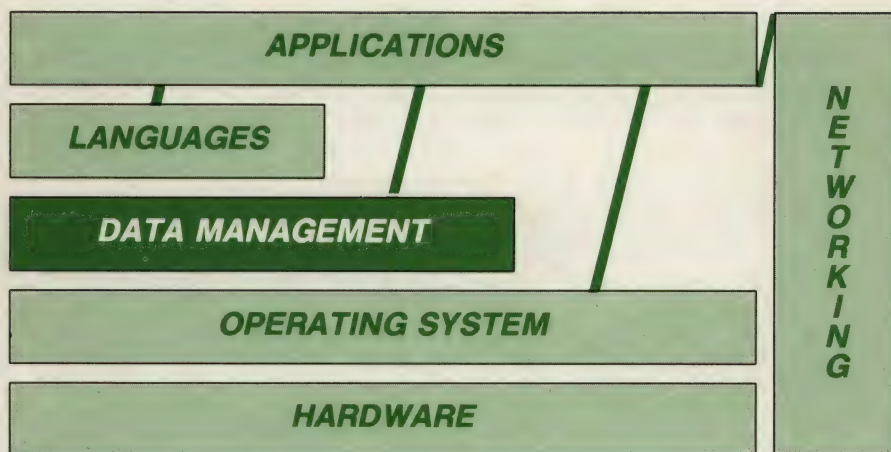
This is well accepted throughout the world and there are significant numbers of certified compilers available.

In order to enhance the portability of programs among PASCAL implementations on X/OPEN systems, the X/OPEN Group has decided to give a uniform definition for certain features designated as implementation-defined in ISO 7185.

- When the required identifiers "input" and "output" occur as program parameters they shall be bound by default to the external system files STDIN and STDOUT respectively.
- There shall be no implementation dependent restrictions on the base-type of a set-type which disallow 0 as the minimal ordinal value and 255 as the maximum ordinal value of that base-type.
- (lazy input) the underlying data transfer action required for a call to the predefined procedure "get" for a textfile (both explicit and where implied by "reset" and "read"), shall be postponed until one of the following events, if any, whichever occurs first :
 - a. access to the buffer-variable of the file
 - b. the buffer-variable of the file is passed as an actual variable parameter to a procedure or function
 - c. a call to the predefined function "eoln" for that file
 - d. a call to the predefined function "eof" for that file

The majority of Pascal compilers also offer extensions beyond the current ISO Standard PASCAL definition. There is little compatibility in these extensions between compilers and developers are warned that use of these extensions may affect the portability of PASCAL programs.

Data Management



6.1 INTRODUCTION

The input/output facilities supported by System V consist only of byte-stream read and write operations on files. No facilities are provided for operating on files as sets of records. This leads to application writers having to make their own arrangements for record handling, resulting in both a multiplication of effort and a proliferation of non-standard methods.

Data Management is a key element in the integration of applications. Applications written in a variety of languages must be able to work on the same basic data in the same form, and data must be passed easily and efficiently between applications.

Addressing these issues, the X/OPEN Group defines interfaces for the creation, management and manipulation of indexed files, generally known as the Indexed Sequential Access Method (ISAM) and for access to relational database management systems, the standard Relational Database Language (SQL).

The availability of these interfaces on X/OPEN systems will not only provide application portability, but will ease and encourage integration.

6.2 INDEXED SEQUENTIAL ACCESS METHOD (ISAM)

The X/OPEN definition for ISAM, which is contained in "INDEXED SEQUENTIAL ACCESS METHOD", is a major subset of the specification of the C-ISAM product, Version 2.10, published by Informix Corporation.

The full specification of C-ISAM contains implementation details specific to that product, in addition to the definition of the interface available to applications. Only the applications interface forms part of the X/OPEN definition; implementation details specific to C-ISAM have been omitted. Indeed, there are alternative implementations available on particular member systems.

6.3 RELATIONAL DATABASE LANGUAGE (SQL)

To reflect the growing significance of Relational Data Base systems, the X/OPEN group has defined application interfaces embedded within high level "host" languages to a relational database management system for a free-standing database.

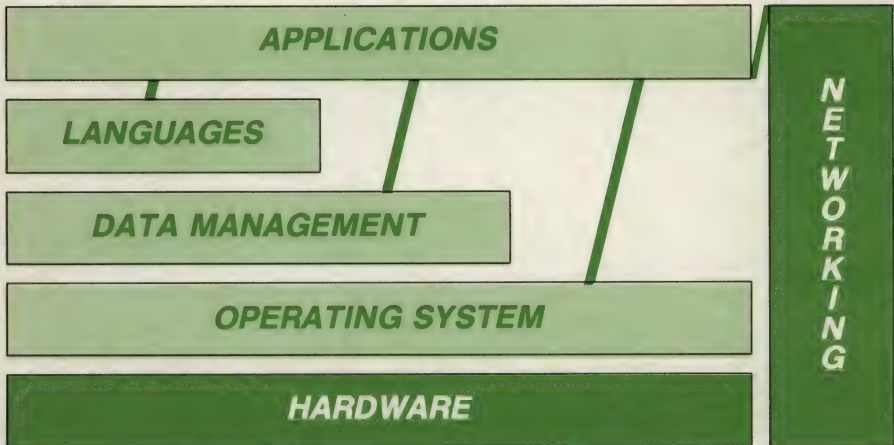
The widely accepted standard for access to relational data base is that defined in the American National Standard document Relational Database Language (SQL) "ANS X3.135-1986".

The X/OPEN definition is based closely on "X3.135-1986" but taking careful account of the capabilities of the leading relational database management systems currently available. The X/OPEN group has worked closely with the vendors of these products throughout.

"X3.135-1986" allows for two levels of compliance, Level 1 and Level 2. Most existing products comply only at Level 1 although it is expected that a significant number of products will have achieved full compliance with Level 2 before the end of 1987. "X3.135-1986" Level 1 SQL is not an adequate definition for application developers, since it leaves too many areas as implementor-defined. In preparing its definition, the X/OPEN group has examined these areas carefully and an agreed X/OPEN approach defined.

The X/OPEN SQL definition is contained in "RELATIONAL DATABASE LANGUAGE (SQL)", and contains a full description of the syntax and semantics of SQL together with a detailed comparison between the X/OPEN definition and the "ANS X3.135-1986" standard.

Source Code Transfer Between Machines



7.1 INTRODUCTION

One of the major problems inhibiting the porting of applications between UNIX system derivatives is that of incompatible media standards and the physical problems of transferring source code in machine readable form.

The X/OPEN Group takes this problem seriously and has agreed common standards for the transfer of source code. Detailed standards are defined in "SOURCE CODE TRANSFER".

Standards are defined for transfer of 5¼" floppy discs and ½" magnetic tape between machines. Because of the different nature of X/OPEN systems, ranging from single user work stations to large mainframes, it is not possible to define formats which are portable across the whole range. Defining standards for both floppy discs and ½" magnetic tape gives the highest practical coverage of systems.

Current differences in the physical recording formats between cartridge tape devices prevents the definition of a standard for this popular medium.

Because of restrictions imposed by existing hardware, some X/OPEN members are not able to support the floppy disc standard.

7.2 FLOPPY DISC STANDARD

As exchange media, the X/OPEN group defines standards for 40 and 80 track floppy discs. It is intended that the prime format should be 80 track, with 40 track retained for compatibility with personal computers. X/OPEN systems equipped only with 80 track disc drives will offer the facility to read 40 track floppy discs by skipping alternate tracks.

7.3 MAGNETIC TAPE

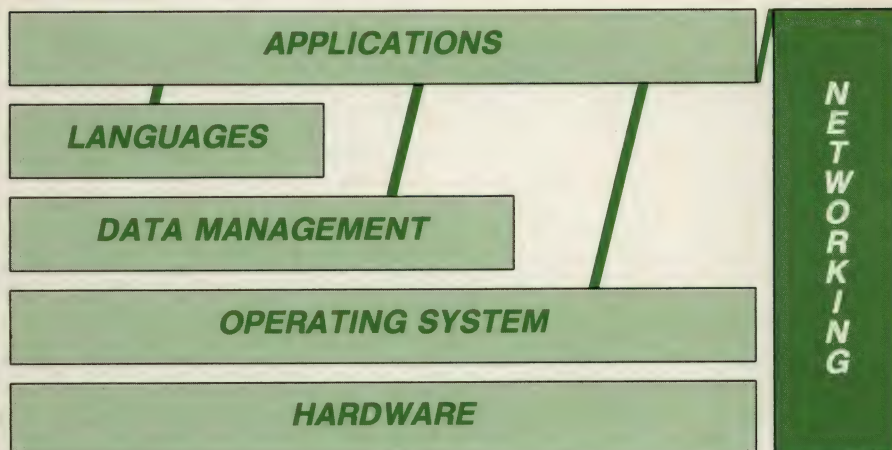
The X/OPEN standard for magnetic tape covers ½" magnetic tape, with a number of different recording formats and densities. The prime format is 9 track Phase Encoded at 1600 bits per inch.

7.4 UTILITIES

"SOURCE CODE TRANSFER" includes the definition of two alternative utilities for the archiving of files to the transfer medium and their subsequent retrieval, *tar* and *cpio*.

In addition, guidelines are given on the use of direct machine to machine connection and the *uucp* utility, as a means of transferring files between X/OPEN systems.

Networking and Communications



8.1 NETWORKING AND COMMUNICATION

The general target for computer data communications is interworking between systems of different types from different suppliers. Future X/OPEN definitions for such open systems interworking can be expected to embrace the ISO OSI standard.

Two services specific to systems supporting the System V Interface have been identified. These are "Generalised Inter Process Communication" (IPC) and "Distributed File System".

Many current commercial applications are supported via interactive transaction processing systems. X/OPEN definitions in this area can be expected to be in line with emerging International Standards.

8.2 OPEN SYSTEMS INTERCONNECTION

For interworking to be possible, systems must have common methods of describing both tasks and data, and must be capable of functioning in a defined manner. To describe interconnection, an architectural model is required. The International Organisation for Standardisation, ISO, has developed the "Reference Model for Open Systems Interconnection" (IS7498). This is often referred to as the "ISO OSI model" or the "ISO 7-layer model". This model sets a framework into which protocol and service standards can be set.

The ISO OSI target is to have a complete set of protocol and service definitions which comply with the 7-layer model and which are internationally agreed and published as ISO standards.

A complete set of ISO OSI standards does not yet exist. Where standards are available, they contain options. For practical systems to be built, it is necessary to have a very clear definition of standards to be adopted and the options to be used.

To provide a clear statement of the standards to be used, a specialist working group has been formed by the 12 major European vendors who propose the technical objectives for "ESPRIT", the "European Strategic Programme for Research into Information Technology". This is called the "Standards Promotion and Application Group, SPAG".

Where ISO standards do not yet exist, interim standards from one of the national or international standards bodies are adopted by SPAG. Such standards are expected to form the basis of future ISO standards. SPAG is not itself a standard making body. Its recommendations will reflect evolving standards.

X/OPEN member companies are committed to the ISO OSI target and the adoption of ISO standards. The group will monitor SPAG recommendations.

X/OPEN intends to define application interfaces for access to OSI services to ensure the portability of applications and library routines.

8.3 GENERALISED INTER-PROCESS COMMUNICATION, IPC

UNIX operating systems provide limited IPC capabilities in the form of "pipes" and "fifos". Kernel extensions within the AT&T System V Interface Definition provide some further IPC mechanisms for the passing of messages between processes in the same memory address space. These extensions were omitted from issue 1 of the X/OPEN definition because it was believed that a much more generalised mechanism for peer to peer communication between processes, either in the same physical machine or in different machines connected via some communications medium is needed.

The X/OPEN group is working on the definition of such a mechanism and but in the short term, it is recognised that there are some applications which need access to such IPC capabilities as currently exist. "XVS INTER-PROCESS COMMUNICATION" gives detailed definitions for

- Message passing between processes.
- Shared memory.
- Semaphores.

It must be recognised that these routines cannot be implemented on all hardware architectures and hence are optional in the X/OPEN definition. However, where the interfaces are supported, the behaviour will be as defined.

8.4 DISTRIBUTED FILE SYSTEM

There is an increasing requirement to be able to access data contained within UNIX File Systems on machines connected together by a local area network from any system on that network. The totality of data accessible in this way can be regarded as a "distributed file system".

The X/OPEN Group regards the way in which local resources are made available to other systems to be a matter of system administration, and does not intend to publish detailed definitions.

The only aspect of such systems which is relevant to the application developer is the behaviour of the system towards applications at run-time.

The characteristics of any distributed file system supported by X/OPEN systems are:

- Access to the distributed file system is via the standard input/output system calls, and is identical for local and remote files.
- No changes are necessary to existing applications. Binary copies of existing applications are able to access a distributed file system, subject to the requirement that the data within a file is in a compatible format.
- Naming of remote items follows the same syntax as for local items and no new naming conventions are required.
- File locking applies across the network.

8.5 DISTRIBUTED TRANSACTION PROCESSING

Many commercial applications require interactive transaction processing facilities and the X/OPEN Group considers the provision of such facilities to be of key importance.

International Standards Organisations have not made the expected progress in this area.

The X/OPEN Group intends to take action to improve this situation.

X/O/P/E/N/

PORTABILITY GUIDE

INDEXED SEQUENTIAL ACCESS
METHOD (ISAM)



Contents

Chapter	1	INTRODUCTION
Chapter	2	ISAM OVERVIEW
Chapter	3	DATA TYPES
	3.1	CHARTYPE
	3.2	INTTYPE AND LONGTYPE
	3.3	FLOATTYPE AND DOUBLETTYPE
Chapter	4	INDEXING
	4.1	INDEX DEFINITION AND MANIPULATION
	4.2	INDEX COMPRESSION
Chapter	5	LOCKING
	5.1	EXCLUSIVE FILE LOCKING
	5.2	MANUAL FILE LOCKING
	5.3	RECORD LEVEL LOCKING
	5.3.1	Automatic Record Locking
	5.3.2	Manual Record Locking
Chapter	6	C PROGRAM EXAMPLES
	6.1	BUILDING A FILE
	6.2	ADDING SECONDARY INDEXES
	6.3	ADDING DATA
	6.4	SEQUENTIAL ACCESS
	6.5	RANDOM ACCESS
	6.6	CHAINING
Chapter	7	EXCEPTION HANDLING
Chapter	8	THE isam.h HEADER FILE
Chapter	9	CALL SPECIFICATIONS
	9.1	RETURN VALUE/EXCEPTION REPORTING
	9.2	isam.h HEADER FILE
	9.3	KEY STRUCTURE
	9.4	RECORD NUMBER OF LAST CALL

© 1982-1985 Informix Corporation.

9.5 CURRENT RECORD POSITION

isaddindex(ISAM)
isbuild(ISAM)
isclose(ISAM)
isdelcurr(ISAM)
isdelete(ISAM)
isdelindex(ISAM)
isdelrec(ISAM)
iserase(ISAM)
isindexinfo(ISAM)
islock(ISAM)
isopen(ISAM)
isread(ISAM)
isrelease(ISAM)
isrename(ISAM)
isrewcurr(ISAM)
isrewrec(ISAM)
isrewrite(ISAM)
isstart(ISAM)
isunlock(ISAM)
iswrcurr(ISAM)
iswrite(ISAM)

Introduction

The input/output facilities supported by the operating system consist only of byte-stream read and write operations on files. No facilities are provided for operating on files as sets of records. This leads to application writers having to make their own arrangements for record handling, resulting in both a multiplication of effort and a proliferation of non-standard methods.

Data management is a key element in the integration of applications. Applications, written in a variety of languages, must be able to work on the same basic data in the same form, and data must be passed easily and efficiently between applications.

As a first step towards addressing these issues, X/OPEN defines an interface for the creation, management and manipulation of indexed files, generally known as the Indexed Sequential Access Method (ISAM). The availability of this interface on X/OPEN systems will not only provide application portability, but will ease and encourage integration.

The X/OPEN ISAM definition is a major sub-set of the specification of the C-ISAM product, version 2.10, published by Informix Corporation of 4100 Bohannon Drive, Menlo Park, California 94025.

The X/OPEN definition omits parts of the C-ISAM specification which are implementation specific. An example is the audit trail facility which is defined in the C-ISAM document without any interfaces for recovery. Internal file formats are given, and the user has to make direct use of these to effect recovery. As alternative implementations may exist, these internal file formats are not part of the X/OPEN standard, and neither, therefore, are the audit trail definitions. (Any use of these facilities on a system that includes them will imply that such applications are not totally portable across X/OPEN systems.)

Version 2.10 of the C-ISAM product introduced new functions, and a set of decimal data types. The new functions were included in the first issue of the X/OPEN specification in the "optional" category, but this limitation has now been removed. The decimal types have been excluded. AT&T have announced that an ISAM capability based on the C-ISAM product will form a part of System V at some stage in the future, and a "Data Management Extension" will be published in a future edition of the System V Interface Definition. The X/OPEN ISAM definition will be further reviewed at that time.

The X/OPEN ISAM Definition is structured as follows:

- Chapter 2 gives an overview of ISAM.
- Chapter 3 describes data types supported by the X/OPEN ISAM definition.
- Chapter 4 describes the definition and manipulation of indexes and techniques for key compression.
- Chapter 5 describes file and record locking techniques to ensure reliable updating in multi-user environments.
- Chapter 6 contains a comprehensive set of C program examples designed to illustrate all the facilities of the ISAM interface.
- Chapter 7 describes the handling of exception conditions.
- Chapter 8 describes the `<isam.h>` header file containing definitions of various macros and symbolic constants.
- Chapter 9 contains detailed specifications of the interfaces in the X/OPEN ISAM definition.

ISAM Overview

The X/OPEN ISAM definition specifies a set of C-language functions that create and manipulate indexed files.

These functions provide for:

- the creation of files and associated primary indexes
- the addition, and deletion, of further indexes
- the opening, closing and deletion of existing files
- the selection of the index to be used for subsequent reading and/or writing of records, and the start point within the file
- the reading, writing and updating of data records
- the locking and unlocking of files and records

When a file is created, two conceptual entities are formed, the container for holding data records and a primary index. The programmer can specify the field, or fields, of each record that is to be used as the primary key for distinguishing the records within the file. As each record is written to the file, an entry is made in the index which stores key value(s) together with the location of the data record in the file. For subsequent reads on the file, individual records are located by searching the index for the required key and using the location stored with it to go straight to the data. Access to a file can be sequential or random.

Indices additional to the primary index can be created. These provide alternative access paths to the same data records by allowing different fields to be used as the keys. The definition puts no limit on the number of alternative indexes that can be created for a file. In an additional index, the same key value is allowed to occur in different records, "duplicates", although a facility is provided to inhibit this on any particular file.

The definition includes the facility to specify index key compression. This allows the density of key storage in an index to be increased, by the use of such techniques as suppression of redundant spaces at the beginning and end of keys and by the elimination of duplicate entries. Only **no compression** and **maximum compression** are fully defined. However, it is recognised that intermediate levels may be provided on any particular member system, and mode values are defined to allow for this. All X/OPEN systems will accept these values to ensure application portability, although the degree of resulting compression may vary.

Facilities are defined for the locking of files and records, to ensure reliable update and access in the multi-user environment. File locking locks out a whole file. It may be exclusive, in that all other accesses to the file are inhibited, or it may be write-only, allowing read accesses to continue. Record level locking may be automatic. In this case it is specified at file open time and a record is automatically locked before it is read, and remains locked until the next function call is completed. Alternatively, it may be manual in that it is actioned as a result of a parameter of a read call.

The following functions are included in the X/OPEN ISAM definition.

FUNCTION NAME	PURPOSE
<i>isaddindex</i> (ISAM)	add index to an ISAM file
<i>isbuild</i> (ISAM)	create an ISAM file
<i>isclose</i> (ISAM)	close an ISAM file
<i>isdelcurr</i> (ISAM)	delete current record
<i>isdelete</i> (ISAM)	delete record specified by primary key
<i>isdelindex</i> (ISAM)	remove index from an ISAM file
<i>isdelrec</i> (ISAM)	delete record specified by record number
<i>iserase</i> (ISAM)	remove an ISAM file
<i>isindexinfo</i> (ISAM)	access file information
<i>islock</i> (ISAM)	lock an ISAM file
<i>isopen</i> (ISAM)	open an ISAM file
<i>isread</i> (ISAM)	read records
<i>isrelease</i> (ISAM)	unlock records
<i>isrename</i> (ISAM)	rename an ISAM file
<i>isrewcurr</i> (ISAM)	rewrite current record
<i>isrewrec</i> (ISAM)	rewrite record specified by record number
<i>isrewrite</i> (ISAM)	rewrite record specified by primary key
<i>isstart</i> (ISAM)	select an index
<i>isunlock</i> (ISAM)	unlock an ISAM file
<i>iswrcurr</i> (ISAM)	write record and set current position
<i>iswrite</i> (ISAM)	write record

The following C-ISAM facilities are not included within the X/OPEN ISAM definition and their use will impede portability:

FUNCTION NAME	PURPOSE
<i>isaudit</i> (ISAM)	performs operations on audit trail
<i>isflush</i> (ISAM)	flushes buffered index pages
<i>issetunique</i> (ISAM)	set unique identifier
<i>isuniqueid</i> (ISAM)	return unique identifier

Also excluded are the decimal data types and associated manipulation routines.

Data Types

The types of data that can be defined and manipulated are described in this chapter. Descriptions of how each data type is stored in files and how each data type must be treated are also included.

The data types for which properly ordered indexes are maintained are type character, 2-byte integers, 4-byte integers, machine float (floating point), and machine double (double precision floating point). The macro definitions used to describe these types are shown below. These definitions can also be found in `<isam.h>`.

<code>CHARTYPE</code>	character
<code>INTTYPE</code>	2-byte integer
<code>LONGTYPE</code>	4-byte integer
<code>FLOATTYPE</code>	machine float
<code>DOUBLETTYPE</code>	machine double

3.1 CHARTYPE

The data type `CHARTYPE` comprises a string of characters, for example a city name or an address.

3.2 INTTYPE AND LONGTYPE

The data types *INTTYPE* and *LONGTYPE* consist of 2- and 4-byte binary signed integer data. Integer data is always stored in files as high/low, most significant byte first, least significant byte last. This storage technique is independent of the form in which integers are stored in the machine on which the system is executing. Therefore, depending on the operating environment, the format of storage for integers in the files may differ from the format of storage for integers stored in executing programs. For this reason, four routines are supplied for the conversion to and from ISAM integer storage format.

The four format conversion routines for integers are:

- | | |
|---------------------|--|
| <i>ldint(p)</i> | returns a machine-format integer; <i>p</i> is a char pointer to the starting byte of format <i>INTTYPE</i> . |
| <i>stint(i, p)</i> | stores a machine-format integer <i>i</i> as format <i>INTTYPE</i> at location <i>p</i> , where <i>p</i> is a char pointer to the first byte of format <i>INTTYPE</i> . |
| <i>ldlong(p)</i> | returns a machine-format long integer; <i>p</i> is a char pointer to the first byte of format <i>LONGTYPE</i> . |
| <i>stlong(l, p)</i> | stores a machine-format long integer <i>l</i> as format <i>LONGTYPE</i> at location <i>p</i> , where <i>p</i> is a char pointer to the first byte of format <i>LONGTYPE</i> . |

These routines are either macros defined in `<isam.h>` or are in the ISAM library.

The typical use for the above routines occurs after a data record has been read into the user buffer. Integer values that are to be used by the user program first have to be converted to machine-usable format by using *ldint* for type *INTTYPE* and *ldlong* for *LONGTYPE*.

Similarly, storage of machine-format integer data requires the use of the *stint* and *stlong* routines.

Note that the formatted integers need not be aligned along word boundaries as do machine-formatted integers.

3.3 FLOATTYPE AND DOUBLETTYPE

The data types *FLOATTYPE* and *DOUBLETTYPE* are the two floating point data types. The data type *FLOATTYPE* is the same as the C data type **float**, while the data type *DOUBLETTYPE* is the same as the C data type **double**. Both data types differ in length and format from machine to machine. There is no difference between the floating point format used and its counterpart in the C language except that floating point numbers may be placed on non-word boundaries. For this reason, four more routines, allow the user to retrieve or replace these non-aligned floating point numbers from their positions in data records. These routines are:

- | | |
|----------------------|---|
| <i>ldfloat(p)</i> | returns a machine-format float ; <i>p</i> is a char pointer to the starting byte of format <i>FLOATTYPE</i> . |
| <i>stfloat(f, p)</i> | stores a machine-format float <i>f</i> at location <i>p</i> , where <i>p</i> is a char pointer to the starting (leftmost) byte of format <i>FLOATTYPE</i> . |
| <i>lddbl(p)</i> | returns a machine-format double ; <i>p</i> is a char pointer to the starting byte of format <i>DOUBLETTYPE</i> . |
| <i>stdbl(d, p)</i> | stores a machine-format double <i>d</i> as format <i>DOUBLETTYPE</i> at location <i>p</i> , where <i>p</i> is a char pointer to the starting (leftmost) byte of format <i>DOUBLETTYPE</i> . |

The use of the floating point load and store routines is analogous to the use of the integer load and store routines.

Indexing

4.1 INDEX DEFINITION AND MANIPULATION

The C language structures that describe an index to any given function call are the *keydesc* and *keypart* structures. These structures are shown below. They are defined in the file `<isam.h>`, which must be included in any program which uses the function calls.

The structure *keydesc* contains the following members:

```
short k-flags; /* compression and duplicates */
short k-nparts; /* number of parts in this key */
struct keypart k-part[NPARTS]; /* each key part */
```

The structure *keypart* contains the following members:

```
short kp-start; /* starting byte of key part */
short kp-leng; /* length in bytes of key part */
short kp-type; /* type of key part */
```

It is the purpose of this chapter to show how to initialise the *keydesc* structure for use with any of the functions that require it as a parameter.

The first sample index to be described here has one part which has the data type of *INTTYPE*. Integers are 2 bytes; therefore, the length of the index is 2 bytes. The index begins in the first byte of the record. No data compression is desired for keys stored in this index. The order of the index is to be ascending (lowest key value to highest key value). Finally, duplicate key values for this index are not to be allowed.

The C program to add the index described above is shown below. It is assumed that the file *myfile* has already been created using the *isbuild*(ISAM) function call.


```

#include <isam.h>

struct keydesc first-key;
int fd;

main()
{
    /* In order to add an index to the file
       "myfile", the file must be opened with
       exclusive access. Therefore, ISEXCLLOCK
       must be arithmetically added to the mode
       parameter. */

    if ((fd = isopen("myfile", ISINOUT+ISEXCLLOCK)) < 0)
    {
        printf("Open error %d on myfile.\n", iserrno);
        exit(1);
    }
    mkfirst-key();
    if (isclose(fd))
    {
        printf("Close error %d on myfile.\n", iserrno);
        exit(1);
    }
}

mkfirst-key()
{
    first-key.k-flags= 0; /* no dups, no compression */
    first-key.k-nparts= 1; /* this index has one part */

    /*The starting byte of an index is always defined
       as the byte offset from the beginning of the
       record. Since this index begins at the begin-
       ning of the record, its byte offset is zero. */

    first-key.k-part[0].kp-start= 0; /* offset is zero */
    first-key.k-part[0].kp-type= INTTYPE; /* data type
                                             is integer */
    first-key.k-part[0].kp-leng= 2; /* 2 byte integer */

    if(isaddindex(fd, &first-key)) /* add the index */
    {
        printf("Error %s iserrno = %d.\n",
               "in adding first-key index: ", iserrno);
    }
}

```

Note that, in the above example, the structure element *k-flags* is initialised to zero. This indicates that no special characteristics are to be attributed to this index. Since *k-flags* is zero, duplicate key values will not be allowed, and no compression will be performed on key values as they are placed in the index.

If duplicate key values were to have been allowed, *k-flags* should have been initialised to *ISDUPS* as in the following statement:

```
/* allow duplicate key values */
first-key.k-flags = ISDUPS;
```

If key value compression had been desired, *k-flags* should have been initialised to *ISDUPS+COMPRESS*. This would allow duplicate key values and would indicate that they be compressed in the index.

```
first-key.k-flags = ISDUPS+COMPRESS;
```

Note, also, that the index defined by the *keydesc* structure *first-key* has only one part. The number of key parts that make up the index is defined by the structure element *k-nparts*, which in the above example is initialised to one.

```
/* this index has one part */
first-key.k-nparts = 1;
```

In the previous example, the index defined had only one part. That part had a data type of *INTTYPE*. However, a particular application could require that a multi-part index be used. Within the *keydesc* structure there exists an array of *keypart* structures. Each *keypart* structure defines one part of the index. It holds the starting byte offset from the beginning of the record, the part's length, and the part's data type. In order for a multi-part index to be described, the user's program must initialize each of these structures to reflect the desired position, length, and data type for each index part.

The structure *keypart* contains the following members:

```
short kp-start;      /* starting byte */
short kp-leng;       /* length in bytes */
short kp-type;       /* type          */
```


In the following example program, a 3-part index is defined. The index consists of a *CHARTYPE* field, a *LONGTYPE*, and another *CHARTYPE* field. It is important to note that the parts of an index need not be contiguous within a record, nor do the parts of an index have to exist in any particular order within the record. However, the maximum number of key parts that can be defined for an index is {NPARTS}, and the total number of bytes within an index cannot exceed {MAXKEYLEN}. There is no limit to the number of keys that can be added to a file.

```
#include <isam.h>

struct keydesc second-key;
int fd;

main()
{
    if ((fd = isopen("myfile", ISINOUT+ISEXCLOCK)) < 0)
    {
        printf("Open error %d on myfile.\n", iserrno);
        exit(1);
    }
    mksecond-key();
    if (isclose(fd))
    {
        printf("Close error %d on myfile.\n", iserrno);
        exit(1);
    }
}

mksecond-key()
{
    /* allow dups, full compression */
    second-key.k-flags      = ISDUPS+COMPRESS;

    /* this index has 3 parts */
    second-key.k-nparts     = 3;

    /* define the first index part */
    second-key.k-part[0].kp-start    = 15;
    second-key.k-part[0].kp-leng     = 8;
    second-key.k-part[0].kp-type     = CHARTYPE;
```

```
/* define the second index part */
second - key.k - part[1].kp - start      = 30;
second - key.k - part[1].kp - leng      = 4;
second - key.k - part[1].kp - type      = LONGTYPE;

/* define the third index part */
second - key.k - part[2].kp - start      = 3;
second - key.k - part[2].kp - leng      = 6;
second - key.k - part[2].kp - type      = CHARTYPE+ISDESC;

if (isaddindex(fd, &second - key))
{
    printf("Error %s iserrno = %d. \n",
           "in adding second - key index: ", iserrno);
}
}
```


4.2 INDEX COMPRESSION

This section discusses key value compression. This allows the density of key storage in an index to be increased by the use of such techniques as suppression of redundant spaces at the beginning and end of keys and the elimination of duplicate entries.

Using these techniques, significant savings can be made in disc space, and substantial improvements obtained in response to random access requests.

Different levels of compression may be available on different machines. To allow for this, the X/OPEN definition is non-specific, but ensures that applications will run across X/OPEN systems without change.

Two levels of space compression are defined: **no compression** and **maximum compression**. The latter calls for the maximum level of space compression available on the machine on which the application is running. The levels apply to each index individually.

In addition, an application can specify whether duplicates are to be allowed for each index.

Duplicates are allowed by setting the value ISDUPS into the *k-flags* field of the *keydesc* structure for a given index, and are inhibited by the value ISNODUPS. (As no default value is defined, either ISDUPS or ISNODUPS must be specified). Space compression is specified by adding the value COMPRESS to ISDUPS or ISNODUPS. All other values in the *k-flags* field are implementation defined, but the X/OPEN system will accept such values as advisory (i.e., applications will not fail, but the level of compression obtained may vary from machine to machine).

Locking

Two levels of locking are defined: file level locking and record level locking. Both are built on the System V lock features. Within these two levels the user can choose from among several methods the one which best suits application requirements.

5.1 EXCLUSIVE FILE LOCKING

File locking may be accomplished in two ways. One method prevents other processes from reading from or writing to a given file. This method is referred to as an exclusive lock and remains in effect from the moment the file is opened, using *isopen*(ISAM) or *isbuild*(ISAM), until the file is closed using *isclose*(ISAM). Exclusive file locking is specified by adding ISEXCLLOCK to the *mode* parameter of the *isopen*(ISAM) or *isbuild*(ISAM) function call.

Exclusive file level locking is not necessary for most situations, but it must be used when an index is being added using *isaddindex*(ISAM) or when an index is being deleted using *isdelindex*(ISAM).

The skeleton program shown below illustrates how exclusive file level locking is done:

```
myfd = isopen("myfile", ISEXCLLOCK+ISINOUT);  
.  
.  
.  
isclose(myfd);
```


5.2 MANUAL FILE LOCKING

Manual file level locking prevents other processes from writing to a given file but allows them to read the locked file. This kind of file level locking is specified by use of the *islock*(ISAM) and *isunlock*(ISAM) function calls. When a file is to be locked in this manner, ISMANULOCK must be added to the *mode* parameter of the *isopen*(ISAM) or *isbuild*(ISAM) call. Later in the program, when locking is desired, *islock*(ISAM) should be called to lock the file. When the file is to be unlocked, *isunlock*(ISAM) should be called. For example:

```
myfd = isopen("myfile", ISMANULOCK+ISINOUT);  
.  
. /* "myfile" is unlocked here */  
.  
islock(myfd);  
.  
. /* "myfile" is locked here */  
.  
isunlock(myfd);  
.  
. /* "myfile" is unlocked here */  
.  
isclose(myfd);
```

5.3 RECORD LEVEL LOCKING

There are two basic types of record level locking: automatic and manual.

Automatic record locking locks a record just before it is read using the *isread*(ISAM) call. It unlocks the record after the next call has completed. Automatic record locking is used when the user wants to lock one record at a time and is unconcerned about when or for how long that record will be locked.

Manual record locking, on the other hand, can lock any number of records. Manual locking locks a record when that record is read using *isread*(ISAM). It unlocks that record, and any other records that are currently locked, when *isrelease*(ISAM) is called. Manual record locking is used when more control is required over when a record, or set of records, is to be locked and unlocked.

Both automatic and manual locking techniques allow other processes to read records locked by the current process, but they may not lock, re-write, or delete them.

5.3.1 Automatic Record Locking

Automatic record locking must be specified when the file is opened. This is done by adding ISAUTOLOCK to the *mode* parameter of the *isopen*(ISAM) or *isbuild*(ISAM) function call. From when the file is opened until it is closed, every record will be locked automatically before it is read. Each record remains locked until the next function call is completed for the current file. Therefore, while using the automatic record locking mechanism, only one record per file may be locked at a given time.

The following illustration shows how automatic record locking is used:

```
myfd = isopen("myfile", ISINOUT+ISAUTOLOCK);  
.  
.  
.  
isread(myfd, myrecord, ISNEXT);  
    /* record locked here */  
    /* before record is read*/  
.  
isrewcurr(myfd, myrecord);  
    /* record unlocked here */  
    /* after completion */  
.  
isclose(myfd);
```


5.3.2 Manual Record Locking

The user's intention to use manual record locking must be specified before any processing takes place. This is done by adding ISMANULOCK to the *mode* parameter of *isopen*(ISAM) or *isbuild*(ISAM) function calls when the file is opened. After the file is open, if the user wishes a record to be locked, ISLOCK must be added to the *mode* parameter of the *isread*(ISAM) function call that is reading that record. Each and every record that is read in this manner remains locked until they are all unlocked by a call of the *isrelease*(ISAM) function. The number of records that may be locked in this manner at any one time is system dependent.

The following illustration shows how a number of records in a particular file are locked and unlocked using manual record locking:

```
myfd = isopen("myfile", ISINOUT+ISMANULOCK);  
.  
.  
.  
isread(myfd, first-record, ISEQUAL+ISLOCK);  
.  
.  
.  
isread(myfd, second-record, ISEQUAL+ISLOCK);  
.  
.  
.  
isread(myfd, third-record, ISEQUAL+ISLOCK);  
.  
.  
.  
isrelease(myfd);  
/* unlock all three records */  
.  
.  
isclose(myfd);
```

C Program Examples

This chapter discusses the creation and manipulation of ISAM files through C language examples. These examples are based on a very simple personnel system. The goal of the personnel system is to keep up-to-date information on employees. This information includes the names, addresses, job titles, and salary histories for all employees.

The personnel system consists of two files, the **employee** file, and the **performance** file. The **employee** file holds personal information about each employee. Each record holds the employee number, name, and address for a single worker. The **performance** file holds information pertaining to each job performance review an employee has had. There is one record for each performance review, job title change, or salary change an employee has had. For every employee record in the **personnel** file there may be many records in the **performance** file. The field definitions for the records in both the **personnel** and **performance** files are shown below:

Employee File Definition

<i>Field Name</i>	<i>Location in Record</i>	
Employee number	0 - 3	LONGTYPE
Last name	4 - 23	CHARTYPE
First name	24 - 43	CHARTYPE
Address	44 - 63	CHARTYPE
City	64 - 83	CHARTYPE

Performance File Definition

<i>Field Name</i>	<i>Location in Record</i>	
Employee number	0 - 3	LONGTYPE
Review date	4 - 9	CHARTYPE
Job rating	10 - 11	CHARTYPE
Salary after review	12 - 19	DOUBLETTYPE
Title after review	20 - 50	CHARTYPE

6.1 BUILDING A FILE

The following C language example creates both the **employee** and the **performance** files. It is important to note that the primary keys must be defined for every file created.

```
#include <isam.h>

#define SUCCESS 0

struct keydesc key;
int cstart, nparts;
int cc, fdemploy, fdperform;

/*
   This program builds the file systems for the
   data files employees and performance.
*/
main()
{
    mkemplkey();
    fdemploy = cc = isbuild("employee",
                          84, &key, ISINOUT+ISEXCLLOCK);
    if (cc < SUCCESS)
    {
        printf("isbuild error %d for %s\n",
              iserrno, "employee file");
        exit(1);
    }
    isclose(fdemploy);

    mkperfkey();
    fdperform = cc = isbuild("perform",
                          49, &key, ISINOUT+ISEXCLLOCK);
    if (cc < SUCCESS)
    {
        printf("isbuild error %d for %s\n",
              iserrno, "performance file");
        exit(1);
    }
    isclose(fdperform);
}
```

```
getfirst()
{
    int cc;

    if (cc = isread(fdemploy, emprec, ISFIRST))
    {
        switch(iserrno)
        {
            case EENDFILE:
                eof = TRUE;
                break;
            default:
                printf("%s error %d\n",
                    "isread ISFIRST", iserrno);
                eof = TRUE;
                return(1);
        }
    }
    return(0);
}

getnext()
{
    int cc;

    if (cc = isread(fdemploy, emprec, ISNEXT))
    {
        switch(iserrno)
        {
            case EENDFILE:
                eof = TRUE;
                break;
            default:
                printf("%s error %d\n",
                    "isread ISNEXT", iserrno);
                eof = TRUE;
                return(1);
        }
    }
    return(0);
}
```



```
mkemplkey()
{
    key.k-flags    = 0;
    key.k-nparts   = 0;
    cstart         = 0;
    nparts         = 0;

    addpart(&key, 4, LONGTYPE);
}

mkperfkey()
{
    key.k-flags    = COMPRESS;
    key.k-nparts   = 0;
    cstart         = 0;
    nparts         = 0;

    addpart(&key, 4, LONGTYPE);
    addpart(&key, 6, CHARTYPE);
}

addpart(keyp, len, type)
register struct keydesc *keyp;
int len;
int type;
{
    keyp->k-part[nparts].kp-start = cstart;
    keyp->k-part[nparts].kp-leng = len;
    keyp->k-part[nparts].kp-type = type;
    keyp->k-nparts = ++nparts;
    cstart += len;
}
```

6.2 ADDING SECONDARY INDEXES

Often the indexes defined to be primary indexes are not adequate for some applications. In the case of this application, two secondary indexes are desirable, an index on the *Last name* field in the **employee** file, and an index on the *Salary* field in the **performance** file. The following program creates these two indexes. It is important to note that while adding indexes, the file must be opened with an exclusive lock. Exclusive file locks are specified in the *mode* parameter of the *isopen*(ISAM) call by initialising that parameter to *ISINOUT+ISEXCLLOCK*. The *ISINOUT* specifies that the file is to be opened for both input and output, and the *ISEXCLLOCK* constant added to *ISINOUT* indicates that the file is to be exclusively locked for the current process and that no other process will be allowed to access this file. Note also that duplicates are to be allowed for both secondary indexes and that *Last name* is to have full compression for its values stored in the index file.

```
#include <isam.h>

#define SUCCESS 0

struct keydesc key;
int cstart, nparts;
int fdemploy, fdperform;

/* This program adds secondary indexes for the last name
   field in the employee file, and the salary field in
   the performance file.
*/

main()
{
    int cc;

    fdemploy = cc = isopen("employee",
                          ISINOUT+ISEXCLLOCK);
    if (cc < SUCCESS)
    {
        printf("isopen error %d %s\n",
              iserrno, "for employee file");
        exit(1);
    }
}
```



```
    mklnamekey();
    cc = isaddindex(fdemploy, &key);
    if (cc != SUCCESS)
    {
        printf("isaddindex error %d for %s\n",
               iserrno, "employee iname key");
        exit(1);
    }
    isclose(fdemploy);

    fdperform = cc = isopen("perform",
                           ISINOUT+ISEXCLLOCK);
    if (cc < SUCCESS)
    {
        printf("isopen error %d for %s\n",
               iserrno, "performance file");
        exit(1);
    }

    mksalkey();
    cc = isaddindex(fdemploy, &key);
    if (cc != SUCCESS)
    {
        printf("isaddindex error %d for %s\n",
               iserrno, "perform sal key");
        exit(1);
    }
    isclose(fdperform);
}
```

```
mklnamekey()
{
    key.k-flags    = ISDUPS + COMPRESS;
    key.k-nparts   = 0;
    cstart         = 4;
    nparts         = 0;

    addpart(&key, 20, CHARTYPE);
}

mksalkey()
{
    key.k-flags    = ISDUPS;
    key.k-nparts   = 0;
    cstart         = 12;
    nparts         = 0;

    addpart(&key, sizeof(double), DOUBLETTYPE);
}

addpart(keyp, len, type)
register struct keydesc *keyp;
int len;
int type;
{
    keyp->k-part[nparts].kp-start = cstart;
    keyp->k-part[nparts].kp-leng = len;
    keyp->k-part[nparts].kp-type = type;
    keyp->k-nparts = ++nparts;
    cstart += len;
}
```


6.3 ADDING DATA

The following program simply adds records to the **employee** file by prompting standard input for values of the fields in the data record. Note that the **employee** file is opened with the **ISOUTPUT** flag as its *mode* parameter.

```
#include <isam.h>
#include <stdio.h>

#define WHOLEKEY 0
#define SUCCESS 0
#define TRUE 1
#define FALSE 0

char emprec[85];
char perfrec[51];
char line[82];
long empnum;

struct keydesc key;
int cstart, nparts;
int fdemploy, fdperform;
int finished = FALSE;

/* This program adds a new employee record to the employee
   file. It also adds that employee's first employee
   performance record to the performance file.
*/
```

```
main()
{
    int cc;

    fdemploy = cc = isopen("employee",
        ISMANULOCK + ISOUTPUT);
    if (cc < SUCCESS)
    {
        printf("isopen error %d %s\n",
            iserrno, "for employee file");
        exit(1);
    }
    fdperform = cc = isopen("perform",
        ISMANULOCK + ISOUTPUT);
    if (cc < SUCCESS)
    {
        printf("isopen error %d %s\n",
            iserrno, "for performance file");
        exit(1);
    }
    getemployee();

    while(!finished)
    {
        addemployee();
        getemployee();
    }
    isclose(fdemploy);
    isclose(fdperform);
}
```



```
getperform()
{
    double new-salary;

    if (empnum == 0)
    {
        finished = TRUE;
        return(0);
    }
    stlong(empnum, perfrec);

    printf("Start Date: ");
    fgets(line, 80, stdin);
    ststring(line, perfrec+4, 6);

    ststring("g", perfrec+10, 1);

    printf("Starting salary: ");
    fgets(line, 80, stdin);
    sscanf(line, "%lf", &new-salary);
    stdbl(new-salary, perfrec+11);

    printf("Title: ");
    fgets(line, 80, stdin);
    ststring(line, perfrec+19, 30);

    printf("\n\n\n");
}

addemployee()
{
    int cc;

    cc = iswrite(fdemploy, emprec);
    if (cc != SUCCESS)
    {
        printf("iswrite error %d %s\n",
               iserrno, "for employee");
        isclose(fdemploy);
        exit(1);
    }
}
```

```
addperform()
{
    int cc;

    cc = iswrite(fdperform, perfrec);
    if (cc != SUCCESS)
    {
        printf("iswrite error %d %s\n",
               iserrno, "for performance");
        isclose(fdperform);
        exit(1);
    }
}

putnc(c,n)
char *c;
int n;
{
    while (n--) putchar(*(c++));
}
```



```
getemployee()
{
    printf("Employee number (enter 0 to exit): ");
    fgets(line, 80, stdin);
    sscanf(line, "%ld", &empnum);
    if (empnum == 0)
    {
        finished = TRUE;
        return(0);
    }
    stlong(empnum, emprec);

    printf("Last name: ");
    fgets(line, 80, stdin);
    ststring(line, emprec+4, 20);

    printf("First name: ");
    fgets(line, 80, stdin);
    ststring(line, emprec+24, 20);

    printf("Address: ");
    fgets(line, 80, stdin);
    ststring(line, emprec+44, 20);

    printf("City: ");
    fgets(line, 80, stdin);
    ststring(line, emprec+64, 20);

    getperform();
    addperform();
    printf("\n\n\n");
}
```

```
/* move NUM sequential characters from SRC to DEST */
ststring(src, dest, num)
char *src;
char *dest;
int num;
{
    int i;

    /* don't move carriage returns or nulls */
    for (i = 1; i <= num && *src != '\n' && src != 0; i++)
        *dest++ = *src++;

    /* pad remaining characters in blanks */
    while (i++ <= num)
        *dest++ = ' ';
}
```


6.4 SEQUENTIAL ACCESS

The next C language example shows how to read a file sequentially. In this particular case the `employee` file is being read in order of the primary key *Employee number*. Since the *Employee number* index is defined as ascending with no duplicate key values allowed, the sequence of records will print from the lowest value of *Employee number* to the highest value of *Employee number*. This will continue until the `isread(ISAM)` call using `ISNEXT` returns the value `[EENDFILE]`, which indicates that the end of file has been reached.

```
#include <isam.h>

#define WHOLEKEY 0
#define SUCCESS 0
#define TRUE 1
#define FALSE 0

char emprec[85];

struct keydesc key;
int cstart, nparts;
int fdemploy, fdperform;
int eof = FALSE;

/* This program sequentially reads through the employee
   file by employee number, printing each record to
   stdout as it goes.
*/
```

```

main()
{
    int cc;

    fdemploy = cc = isopen("employee",
        ISINPUT+ISAUTOLOCK);
    if (cc < SUCCESS)
    {
        printf("isopen error %d %s\\",
            iserrno, "for employee file");
        exit(1);
    }
    mkemplkey();
    cc = isstart(fdemploy, &key, WHOLEKEY, emprec, ISFIRST);
    if (cc != SUCCESS)
    {
        printf("isstart error %d\\n", iserrno);
        isclose(fdemploy);
        exit(1);
    }
    getfirst();
    while (!eof)
    {
        showemployee();
        getnext();
    }
    isclose(fdemploy);
}

showemployee()
{
    printf("Employee number: %ld", ldlong(emprec));
    printf("\\nLast name: ");      putnc(emprec+4, 20);
    printf("\\nFirst name: ");    putnc(emprec+24, 20);
    printf("\\nAddress: ");       putnc(emprec+44, 20);
    printf("\\nCity: ");          putnc(emprec+64, 20);
    printf("\\n\\n\\n");
}

putnc(c, n)
char *c;
int n;
{
    while (n--) putchar(*(c++));
}

```



```
getfirst()
{
    int cc;

    if (cc = isread(fdemploy, emprec, ISFIRST))
    {
        switch(iserrno)
        {
            case EENDFILE:
                eof = TRUE;
                break;
            default:
                printf("isread ISFIRST error %d\n",
                    iserrno);
                eof = TRUE;
                return(1);
        }
    }
    return(0);
}

getnext()
{
    int cc;

    if (cc = isread(fdemploy, emprec, ISNEXT))
    {
        switch(iserrno)
        {
            case EENDFILE:
                eof = TRUE;
                break;
            default:
                printf("isread ISNEXT error %d\n",
                    iserrno);
                eof = TRUE;
                return(1);
        }
    }
    return(0);
}
```

```
mkemplkey()
{
    key.k-flags = 0;
    key.k-nparts = 0;
    cstart = 0;
    nparts = 0;

    addpart(&key, 4, LONGTYPE);
}

addpart(keyp, len, type)
register struct keydesc *keyp;
int len;
int type;
{
    keyp->k-part[nparts].kp-start = cstart;
    keyp->k-part[nparts].kp-leng = len;
    keyp->k-part[nparts].kp-type = type;
    keyp->k-nparts = ++nparts;
    cstart += len;
}
```


6.5 RANDOM ACCESS

The following program is an example of how random access to a file can be accomplished. This program interactively retrieves an employee number from standard input, searches for it in the **employee** file, and prints the results of its search to standard output.

Note that the `ISEQUAL` constant is used to specify the read mode to `isread(ISAM)` in the C function called `reademp`. If no record corresponding to the value entered by the user is found for *Employee number*, a condition code of `[ENOREC]` is returned by `isread(ISAM)`. It is the responsibility of the C programmer to handle that return code in an appropriate manner. If `[ENOREC]` is returned, the record buffer sent as the record parameter to the `isread(ISAM)` call will not have been changed (that is, no record will have been read).

```
#include <isam.h>
#include <stdio.h>

#define WHOLEKEY 0
#define SUCCESS 0
#define TRUE 1
#define FALSE 0

char emprec[83];
char line[80];
long empnum;

struct keydesc key;
int cstart, nparts;
int fdemploy, fdperform;
int eof = FALSE;

/*
   This program interactively retrieves an employee's employee
   number from stdin, searches for it in the employee file,
   and prints the employee record that has that value as its
   employee number field.
*/
```

```
main()
{
    int cc;

    fdemploy = cc = isopen("employee",
        ISINPUT+ISAUTOLOCK);
    if (cc < SUCCESS)
    {
        printf("isopen error %d %s\n",
            iserrno, "for employee file");
        exit(1);
    }
    mkemplkey();
    getempnum();
    while (empnum != 0)
    {
        if (reademp() == SUCCESS) showemployee();
        getempnum();
    }
    isclose(fdemploy);
}

getempnum()
{
    printf("Enter the employee number (0 to quit): ");
    fgets(line, 80, stdin);
    sscanf(line, "%ld", &empnum);
    stlong(empnum, emprec);
}

showemployee()
{
    printf("Employee number: %ld", ldlong(emprec));
    printf("\nLast name: ");      putnc(emprec+4, 20);
    printf("\nFirst name: ");     putnc(emprec+24, 20);
    printf("\nAddress: ");        putnc(emprec+44, 20);
    printf("\nCity: ");           putnc(emprec+64, 20);
    printf("\n\n");
}
```



```
putnc(c, n)
char *c;
int n;
{
    while (n-- > 0) putchar(*c++);
}

reademp()
{
    int cc;

    cc = isread(fdemploy, emprec, ISEQUAL);
    if (cc != SUCCESS)
    {
        switch (iserrno)
        {
            case EENDFILE:
                eof = TRUE;
                break;
            default:
                printf("isread ISEQUAL error %d\n",
                    iserrno);
                eof = TRUE;
                return(1);
        }
    }
    return(0);
}

mkemplkey()
{
    key.k_flags = 0;
    key.k_nparts = 0;
    cstart = 0;
    nparts = 0;

    addpart(&key, 4, LONGTYPE);
}
```

```
addpart(keyp, len, type)
register struct keydesc *keyp;
int len;
int type;
{
    keyp->k-part[nparts].kp-start = cstart;
    keyp->k-part[nparts].kp-leng = len;
    keyp->k-part[nparts].kp-type = type;
    keyp->k-nparts = ++nparts;
    cstart += len;
}
```


6.6 CHAINING

The following example shows how to chain to a record that is the last record in a chain of associated records, illustrating how the performance records appear logically by the primary key. The primary index is a composite index made up of the *Employee number* and *Review date*.

Emp. No.	Review Date	Job New Rating	New Salary	Title
1	790501	g	20,000	PA
1	800106	g	23,000	PA
1	800505	f	24,725	PA
2	760301	g	18,000	JP
2	760904	g	20,700	PA
2	770305	g	23,805	PA
2	770902	g	27,376	SPA
3	800420	f	18,000	JP
4	800420	f	18,000	JP

```
#include <isam.h>
#include <stdio.h>

#define WHOLEKEY 0
#define SUCCESS 0
#define TRUE 1
#define FALSE 0

char perfrec[51];
char operfrec[51];
char line[81];
long empnum;
double new - salary, old - salary;

struct keydesc key;
int cstart, nparts;
int fdemploy, fdperform;
int finished = FALSE;

/* This program interactively reads data from stdin and adds
performance records to the "perform" file. Depending on
the rating given the employee on job performance, the
following salary increases are placed in the salary field
of the performance file.

    rating      percent increase
    p (poor)    0.0 %
    f (fair)    7.5 %
    g (good)    13.5 %
*/
```



```
main()
{
    int cc;

    fdperform = cc = isopen("perform",
        ISINOUT+ISAUTOLOCK);
    if (cc < SUCCESS)
    {
        printf("isopen error %d %s\n",
            iserrno, "for performance file");
        exit(1);
    }
    mkperfkey();
    getperformance();
    while (!finished)
    {
        if (get-old-salary())
        {
            finished = TRUE;
        }
        else
        {
            addperformance();
            getperformance();
        }
    }
    isclose(fdperform);
}

addperformance()
{
    int cc;

    cc = iswrite(fdperform, perfrec);
    if (cc != SUCCESS)
    {
        printf("iswrite error %d\n", iserrno);
        isclose(fdperform);
        exit(1);
    }
}
```

```
getperformance()
{
    printf("Employee number (enter 0 to exit): ");
    fgets(line, 80, stdin);
    sscanf(line, "%ld", &empnum);
    if (empnum == 0)
    {
        finished = TRUE;
        return(0);
    }
    stlong(empnum, perfrec);

    printf("Review Date: ");
    fgets(line, 80, stdin);
    ststring(line, perfrec+4, 6);

    printf("Job rating (p = poor, f = fair, g = good): ");
    fgets(line, 80, stdin);
    ststring(line, perfrec+10, 1);

    printf("Salary After Review: ");
    printf("(Sorry, you don't get to add this)\n");
    new_salary = 0.0;
    stdbl(new_salary, perfrec+11);
    printf("Title After Review: ");
    fgets(line, 80, stdin);
    ststring(line, perfrec+19, 30);

    printf("\n\n\n");
}
```



```
get-old-salary()
{
    int mode, cc;

    /* get employee id no. */
    bytecpy(perfrec, operfrec, 4);

    /* largest possible date */
    bytecpy("999999", operfrec+4, 6);

    cc = isstart(fdperform, &key,
                WHOLEKEY, operfrec, ISGTEQ);
    if (cc != SUCCESS)
    {
        switch(iserrno)
        {
            case ENOREC:
            case EENDFILE:
                mode = ISLAST;
                break;
            default:
                printf("isstart error %d\n",
                    iserrno);
                return(1);
        }
    }
    else
    {
        mode = ISPREV;
    }
    cc = isread(fdperform, operfrec, mode);
    if (cc != SUCCESS)
    {
        printf("isread error %d %s\n",
            iserrno, "in get-old-salary");
        return(1);
    }
}
```

```

if (cmpnbytes(perfrec, operfrec, 4))
{
    printf("%s for employee number %ld\n",
           "No performance record", llong(operfrec));
    return(1);
}
else
{
    printf("\nPerformance record found.\n\n");
    old-salary = new-salary = lddbl(operfrec+11);
    printf("Rating: ");
    switch(*(perfrec+10))
    {
        case 'p':
            printf("poor\n");
            break;
        case 'f':
            printf("fair\n");
            new-salary *= 1.075;
            break;
        case 'g':
            printf("good\n");
            new-salary *= 1.15;
            break;
    }
    stdbl(new-salary, perfrec+11);
    printf("Old salary was %f\n", old-salary);
    printf("New salary is %f\n", new-salary);
    return(0);
}
}

bytecpy(src,dest,n)
register char *src;
register char *dest;
register int n;
{
    while (n-- > 0)
    {
        *dest++ = *src++;
    }
}

```



```

cmpnbytes(byte1, byte2, n)
register char *byte1, *byte2;
register int n;
{
    if (n <= 0) return(0);
    while (*byte1 == *byte2)
    {
        if (--n == 0) return(0);
        ++byte1;
        ++byte2;
    }
    return((( *byte1 & BYTEMASK) <
            ( *byte2 & BYTEMASK)) ? -1 : 1);
}

```

```

mkperfkey()
{
    key.k_flags = COMPRESS;
    key.k_nparts = 0;
    cstart = 0;
    nparts = 0;

    addpart(&key, 4, LONGTYPE);
    addpart(&key, 6, CHARTYPE);
}

```

```

/* move NUM sequential characters from SRC to DEST */
ststring(src, dest, num)
char *src;
char *dest;
int num;
{
    int i;

    /* don't move carriage returns or nulls */
    for (i = 1; i <= num && *src != '\n' && src != 0; i++)
        *dest++ = *src++;

    /* pad remaining characters in blanks */
    while (i++ <= num)
        *dest++ = ' ';
}

```

```
addpart(keyp, len, type)
register struct keydesc *keyp;
int len;
int type;
{
    keyp->k-part[nparts].kp-start = cstart;
    keyp->k-part[nparts].kp-leng = len;
    keyp->k-part[nparts].kp-type = type;
    keyp->k-nparts = ++nparts;
    cstart += len;
}
```


Exception Handling

Calls to ISAM functions generally return a value of 0 to indicate success or -1 to indicate some kind of exception. In the latter case, the global integer *iserrno* and the global characters *isstat1* and *isstat2* are set to meaningful values to define the nature of the condition. When testing return values in *iserrno*, it is recommended that the symbolic names defined in `<isam.h>` be used, rather than absolute values.

ISAM codes indicate the following:

NAME	No.	TEXT	STATUS BYTE 1	STATUS BYTE 2
[EDUPL]	100	An attempt was made to add a duplicate value to an index via <i>iswrite</i> , <i>isrewrite</i> , <i>isrewcurr</i> or <i>isaddindex</i> .	2	2
[ENOTOPEN]	101	An attempt was made to perform some operation on a file that was not previously opened using the <i>isopen</i> call.	9	0
[EBADARG]	102	One of the arguments of the call is not within the range of acceptable values for that argument.	9	0
[EBADKEY]	103	One or more of the elements that make up the key description is outside of the range of acceptable values for that element.	9	0
[ETOOMANY]	104	The maximum number of files that may be open at one time would be exceeded if this request were processed.	9	0
[EBADFILE]	105	The format of the file has been corrupted.	9	0
[ENOTEXCL]	106	In order to add or delete an index, the file must have been opened with exclusive access.	9	0
[ELOCKED]	107	The record requested by this call cannot be accessed because it has been locked by another user.	9	0
[EKEXISTS]	108	An attempt was made to add an index that has been defined previously.	9	0
[EPRIMKEY]	109	An attempt was made to delete the primary key value. The primary key may not be deleted by the <i>isdelindex</i> call.	9	0
[EENDFILE]	110	The beginning or end of file was reached.	1	0

Exception Handling

NAME	No.	TEXT	STATUS BYTE 1	STATUS BYTE 2
[ENOREC]	111	No record could be found that contained the requested value in the specified position.	2	3
[ENOCURR]	112	This call must operate on the current record. One has not been defined.	2	1
[EFLOCKED]	113	The file is exclusively locked by another user.	9	0
[EFNAME]	114	The file name is too long.	9	0

Two bytes are used to hold status information after calls. They are related in the following way. The first byte holds status information of a general nature, such as success or failure of a call. The second byte contains more specific information that has meaning based on the status code in byte one. The values of the status bytes are:

Byte One

- 0 Successful Completion
- 1 End of File
- 2 Invalid Key
- 3 System Error
- 9 User Defined Errors

Byte Two

When status key one is:	Status key two indicates:	
0 - 9	0	No further information is available
0	2	Duplicate key indicator <ul style="list-style-type: none"> - After a <i>isread</i>(ISAM) this indicates that the key value for the current key is equal to the value of that same key in the next record. - After a <i>isread</i>(ISAM) or <i>isrewrite</i>(ISAM) this indicates that the record just written created a duplicate key value for at least one alternate record key for which duplicates are allowed.
2	1	The primary key value has been changed by the program between the successful execution of a <i>isread</i> (ISAM) statement and the execution of the next <i>isrewrite</i> (ISAM) statement.
	2	An attempt has been made to write or rewrite a record that would create a duplicate key in an indexed file.
	3	No record with the specified key can be found.
	4	An attempt has been made to write beyond the externally defined boundaries of an indexed file.
9		The value of status key two is defined by the user.

The isam.h Header File

This chapter defines the contents of the header file `<isam.h>`. The file contains definitions that are used for the mode arguments and also definitions of structures that are used in the calls.

Definitions that specify limits in the above table give the limit that can be assumed by applications for full portability across X/OPEN machines. There will be at least that number on a given system, although there may in fact be more.

For example, `{NPARTS}` gives the maximum number of key parts, and it is set to 8. This means that all X/OPEN systems will allow at least 8 key parts. It also means that, for full portability, an application should not require more than this number. A particular X/OPEN machine may allow more than 8 and, on that system, the definition will be set to a higher value. However, applications relying on this higher value are not guaranteed to be portable.

The isam.h header file:

```
#define CHARTYPE      0
#define CHARSIZE      1

#define INTTYPE       1
#define INTSIZE       2

#define LONGTYPE      2
#define LONGSIZE      4

#define DOUBLETTYPE   3
#define DOUBLESIZE    (sizeof(double))

#define FLOATTYPE     4
#define FLOATSIZE     (sizeof(float))

#define MAXTYPE       5
#define ISDESC        0x80          /* add to make */
                                   /* descending type */
#define TPEMASK       0x7F          /* type mask */

#define BYTEMASK      0xFF          /* mask for one byte */
#define BYTESHFT      8            /* shift for one byte */

#define ldint(p) ((short)(((p)[0]<<BYTESHFT)+((p)[1]&BYTEMASK)))
#define stint(i,p) ((p)[0]=(i)>>BYTESHFT,(p)[1]=(i))
long ldlong();

double ldfloat();
double lddbl();

#define ISFIRST       0            /* first record */
#define ISLAST        1            /* last record */
#define ISNEXT        2            /* next record */
#define ISPREV        3            /* previous record */
#define ISCURR        4            /* current record */
#define ISEQUAL       5            /* equal value */
#define ISGREAT       6            /* greater value */
#define ISGTEQ        7            /* >= value */

/* isread lock modes */
#define ISLOCK        0x100        /* lock record before reading */
```


The isam.h Header File

```
/* isopen, isbuild lock modes */
#define ISAUTOLOCK    0x200    /* automatic record lock    */
#define ISMANULOCK    0x400    /* manual record lock      */
#define ISEXCLLOCK    0x800    /* exclusive isam file lock */

#define ISINPUT       0        /* open for input only      */
#define ISOUTPUT      1        /* open for output only     */
#define ISINOUT       2        /* open for input and output */

#define MAXKEYSIZE    120      /* max number of bytes in key */
#define NPARTS        8        /* max number of key parts   */

struct keypart
{
    short kp-start;    /* starting byte of key part */
    short kp-leng;     /* length in bytes           */
    short kp-type;     /* type of key part          */
};

struct keydesc
{
    short k-flags;     /* flags                      */
    short k-nparts;    /* number of parts in key    */
    struct keypart
    k-part[NPARTS];    /* each key part             */

/* the following is for internal use only */
    short k-len;       /* length of whole key       */
    long k-rootnode;   /* pointer to rootnode       */
};

#define k-start      k-part[0].kp-start
#define k-leng       k-part[0].kp-leng
#define k-type       k-part[0].kp-type

#define ISNODUPS      000      /* no duplicates and no */
                                /* compression allowed */
#define ISDUPS        001      /* duplicates allowed */
#define COMPRESS      016      /* full compression */
```

```
struct dictinfo
```

```
{
    short di-nkeys;      /* number of keys defined */
    short di-reclsize;    /* data record size */
    short di-idxsize;     /* index record size */
    long di-nrecords;     /* number of records */
};                       /* in file */
```

```
#define EDUPL          100  /* duplicate record */
#define ENOTOPEN        101  /* file not open */
#define EBADARG          102  /* illegal argument */
#define EBADKEY          103  /* illegal key desc */
#define ETOOMANY         104  /* too many files open */
#define EBADFILE         105  /* bad ISAM file format */
#define ENOTEXCL         106  /* non-exclusive access */
#define ELOCKED          107  /* record locked */
#define EKEXISTS          108  /* key already exists */
#define EPRIMKEY          109  /* is primary key */
#define EENDFILE         110  /* end/begin of file */
#define ENOREC           111  /* no record found */
#define ENOCURR          112  /* no current record */
#define EFLOCKED         113  /* file locked */
#define EFNAME           114  /* file name too long */
#define EBADMEM          116  /* can't alloc memory */
#define EBADCOLL         117  /* bad custom collating */
```

```
/*
 * For system call errors
 * iserrno = errno (system error code 1-99)
 * iserrio = IO-call + IO-file
 * IO-call = what system call
 * IO-file = which file caused error
 */
```

```
#define IO-OPEN          0x10  /* open() */
#define IO-CREA          0x20  /* creat() */
#define IO-SEEK          0x30  /* lseek() */
#define IO-READ          0x40  /* read() */
#define IO-WRIT          0x50  /* write() */
#define IO-LOCK          0x60  /* locking() */
#define IO-IOCTL         0x70  /* ioctl() */
```

```
extern int iserrno;      /* isam error return code */
```

The isam.h Header File

```
extern int iserrio;      /* system call error code   */
extern long isrecnum;    /* record number of last call */
extern char isstat1;     /* cobol status characters    */
extern char isstat2;
```

```
/* error message usage:
```

```
 *
 * if (iserrno >= 100 && iserrno < is-nerr)
 *     printf("ISAM error %d: %s\n",
 *           iserrno, is-errlist[iserrno-100]);
 */
```


Call Specifications

This chapter contains detailed descriptions of the X/OPEN ISAM functions. The following general notes apply throughout.

9.1 RETURN VALUE/EXCEPTION REPORTING

Most calls return either a 0 or a -1 as the value of the function and set the global integer *iserrno* either to 0 or to an error indicator. In the case of *isbuild*(ISAM) or *isopen*(ISAM), the return value will be a legal file descriptor or a -1. A -1 indicates that an error has occurred, and *iserrno* has been set. Also, the global characters *isstat1* and *isstat2* are set for the convenience of integration with COBOL. See Chapter 7, "Exception Handling", for more information.

9.2 <isam.h> HEADER FILE

Some parameters in this chapter are declared to be structure types that are defined in the <isam.h> header file. Also defined are symbolic values.

9.3 KEY STRUCTURE

The structures *keydesc* and *keypart*, also defined in `<isam.h>`, are used for index definition and are further explained below:

The structure *keydesc* contains the following members:

```
short k-flags;    /* flags */
short k-nparts;   /* number of parts in key */
struct keypart k-part[NPARTS]; /* each key part */
```

The structure *keypart* contains the following members:

```
short kp-start;   /* starting byte of key part */
short kp-leng;    /* length in bytes */
short kp-type;    /* type of key part */
```

In the *keydesc* structure, the integer *k-flags* is used to hold duplicate and compression information for the index that is being added, deleted, or selected. The symbolic values that are defined in `<isam.h>` should be used to indicate the compression techniques that are desired. If more than one feature is specified, the values are logically ORed together. The meaning of these symbolic values is:

ISDUPS	Duplicate values are allowed for this index.
ISNODUPS	No duplicates.
COMPRESS	Full compression for this index.

One of ISDUPS and ISNODUPS must be specified. Compression is requested by the addition of COMPRESS.

k-nparts is an integer that indicates how many parts make up the index. These parts must be described in the *k-part* array of *keypart* structures. A *keypart* structure defines each part of the index individually. The number of elements in the *k-part* array should be equal to the integer value in *k-nparts*.

The elements in the *keypart* structure are used as follows. *kp-start* indicates the starting byte of the key part that is being defined. *kp-leng* is a count of the number of bytes in the part, and *kp-type* designates the data type of the part. The types allowed are defined in the header file, `<isam.h>`, see Chapter 8, "The isam.h Header File". If this part of the key is in descending order, the type constant should be ORed to the ISDESC constant (defined in `<isam.h>`). For more information about creating and manipulating indexes, see Chapter 4, "Indexing".

9.4 RECORD NUMBER OF LAST CALL

Isrecnum is a 4-byte field that is set following the successful completion of all record-based calls. It identifies, in an implementation-dependent, shorthand way, the record just referenced. This returned value may be used in input to the *isdelrec*(ISAM), *isread*(ISAM), and *isrewrec*(ISAM) calls to perform optimised deletes, reads, and updates. If used to perform sequential processing, the records will be read according to their physical layout on disc, and not according to any logical key order. Note that as the actual value returned is implementation-dependent, the user should not attempt to interpret its actual value, as this could compromise portability.

The following calls set *isrecnum*:

<i>isdelcurr</i> (ISAM)	<i>isdelete</i> (ISAM)	<i>isdelrec</i> (ISAM)	<i>isread</i> (ISAM)
<i>isrewcurr</i> (ISAM)	<i>isrewrec</i> (ISAM)	<i>isrewrite</i> (ISAM)	<i>isstart</i> (ISAM)
<i>iswrcurr</i> (ISAM)	<i>iswrite</i> (ISAM)		

9.5 CURRENT RECORD POSITION

The current record position should not be confused with *isrecnum* (see above). The current record position allows sequential processing to be performed according to a logical key order. The mode parameters ISNEXT and ISPREV are thus always relative to this value, while ISCURRE indicates that this (the current) record should be read. If the current record is deleted (by using *isdelcurr*(ISAM)), the current record position will not change and will continue to indicate the now deleted record. The current record may be rewritten directly using *isrewcurr*(ISAM).

The current record position is set after the successful completion of the following calls:

<i>isopen</i> (ISAM)	<i>isread</i> (ISAM)	<i>isstart</i> (ISAM)	<i>iswrcur</i> (ISAM)
----------------------	----------------------	-----------------------	-----------------------

and used in input to:

<i>isdelcurr</i> (ISAM)	<i>isread</i> (ISAM)	<i>isrewcurr</i> (ISAM)
-------------------------	----------------------	-------------------------

NAME

isaddindex — add index to an ISAM file

SYNTAX

```
isaddindex(isfd, keydesc)
int isfd;
struct keydesc *keydesc;
```

DESCRIPTION

Isaddindex is used to add an index to an ISAM file. The index will be built for the file indicated by the *isfd* parameter and will be defined according to the information in the *keydesc* structure. This call will execute only if the file has been opened for exclusive access.

There is no limit to the number of indexes that may be added through the *isaddindex* call. However, the maximum number of parts that may be defined for an index is {NPARTS}, and the maximum number of bytes that can exist in an index is {MAXKEYSIZE} (see Chapter 8, "The isam.h Header File").

Use of this call and index use in general are explained in Chapter 4, "Indexing".

NAME

isbuild — create an ISAM file

SYNTAX

```
isbuild(filename, recordlength, keydesc, mode)
char *filename;
int recordlength;
struct keydesc *keydesc;
int mode;
```

DESCRIPTION

Isbuild is used to create an ISAM file. Depending on the particular implementation, this call will create and initialise appropriate disc structures to contain data and indexes.

After *isbuild* has completed successfully, the file will remain open for further processing. The *isbuild* function returns a file descriptor.

The *filename* parameter should contain a null-terminated character string which is at least four characters shorter than the longest legal operating system file name.

The *recordlength* parameter is the length of the record. Its value is the sum of the number of bytes in each field of the record. See Chapter 3, "Data Types" for the length of each data type.

All ISAM files are required formally to have a primary index. The *keydesc* parameter of this call is used to specify the structure of the primary index. However, setting *k-nparts* = 0 means that there is actually no primary key. Additional indexes may be added later using *isaddindex*. See Chapter 4, "Indexing" and Chapter 6, "C Program Examples" for more details on key definition and use.

The *mode* parameter is used to specify locking information. The user has three options: manual, automatic, or exclusive. Selecting the manual option indicates that the user wishes to be responsible for locking records at the appropriate times using either the *islock*(ISAM) and *isunlock*(ISAM) calls or the ISLOCK mode flag of the *isread*(ISAM) call and the *isrelease*(ISAM) function call. Selecting automatic locking indicates that the user wishes to lock each record at the time it is read and unlock each record after the next function call is made. Selection of exclusive locking will deny file access to anyone other than this process. More information about locking can be found in Chapter 5, "Locking". The *mode* is specified by using the define macros that are found in the header file <isam.h>, for which a complete listing can be found in Chapter 8, "The isam.h Header File".

Modes that are used in the *isbuild* call are:

one of these,	added arithmetically to one of these:
ISEXCLLOCK	ISINPUT
ISMANULOCK	ISOUTPUT
ISAUTOLOCK	ISINOUT

NAME

isclose — close an ISAM file

SYNTAX

```
isclose(isfd)  
int isfd;
```

DESCRIPTION

isclose is used to close an ISAM file. Any locks that are held for the file by the process issuing the *isclose* call are released.

NOTE: it is mandatory to close ISAM files after processing has finished. Failure to do so could cause unpredictable results.

NAME

isdelcurr — delete current record

SYNTAX

```
isdelcurr(isfd)  
int isfd;
```

DESCRIPTION

Isdelcurr differs from *isdelete*(ISAM) in that it deletes the current record from the file, rather than the record indicated by the primary key. The appropriate values will be deleted from each index that is defined. This call is useful when the primary key is not unique and the record cannot be located and deleted in one call. *Isrecnum* is set to indicate the current record, (the record just deleted), whose position is left unchanged.

NAME

isdelete — delete record specified by primary key

SYNTAX

```
isdelete(isfd, record)
int isfd;
char *record;
```

DESCRIPTION

isdelete deletes the record specified by a unique primary key from the file indicated by *isfd*. The appropriate values will also be deleted from each index. If the primary index allows duplicates, then *isread*(ISAM) and *isdelcurr*(ISAM) should be used instead. *Isreclnum* is set to indicate the record just deleted, while the current record position is left unchanged.

NAME

isdelindex — remove index from an ISAM file

SYNTAX

```
isdelindex(isfd, keydesc)
int isfd;
struct keydesc *keydesc;
```

DESCRIPTION

Isdelindex is used to remove an existing index. The index will be removed from the file indicated by *isfd*. The index to be removed will be defined by the information in the *keydesc* structure. All indexes may be deleted except the primary index. Attempts to delete the primary index will cause an error code (-1) to be returned and the *iserrno* global integer to be set. This call will execute only if the file has been opened for exclusive access.

NAME

isdelrec — delete record specified by record number

SYNTAX

isdelrec(isfd, recnum)

int isfd;

long recnum;

DESCRIPTION

Isdelrec differs from *isdelete*(ISAM) in that it deletes the record specified by *recnum* from the file indicated by *isfd*, rather than the record indicated by the primary key. The appropriate values will be deleted from each index that is defined. *Recnum* must be a previously obtained *isrecnum* value. This call will set *isrecnum* to the value of *recnum*, while the current record position is left unchanged.

NAME

iserase — remove an ISAM file

SYNTAX

iserase(filename)
char *filename;

DESCRIPTION

iserase will remove the file specified by *filename*.

NAME

isindexinfo — access file information

SYNTAX

```
isindexinfo(isfd, buffer, number)
int isfd;
struct keydesc *buffer;
/* buffer may be a pointer to */
/* a dictinfo structure instead. */
int number;
```

DESCRIPTION

Isindexinfo gives the caller access to information about the file, such as information about the defined indexes, their location within the record, their length, and whether duplicate values are allowed.

Information about a particular index is obtained by specifying the number of the index using the *number* parameter. General information such as the number of indexes, index record size, and data record size is obtained by calling *isindexinfo* with the *number* parameter set to 0 and reading the *buffer* into a structure of type *dictinfo*.

The *buffer* parameter can contain information in the format of either *keydesc* or *dictinfo* depending on whether the *number* parameter is positive or 0, respectively. As indexes are added and deleted, the number of a particular index may vary. To ensure review of all indexes, loop over the number of indexes indicated in *dictinfo* (see structure definitions in Chapter 8, "The isam.h Header File").

NAME

islock — lock an ISAM file

SYNTAX

```
islock(isfd)  
int isfd;
```

DESCRIPTION

islock will lock the entire file that is specified by *isfd*. More discussion of locking can be found in Chapter 5, "Locking".

NAME

isopen — open an ISAM file

SYNTAX

```
isopen(filename, mode)
char *filename;
int mode;
```

DESCRIPTION

Isopen is used to open an ISAM file for processing. The function will return the file descriptor that should be used in subsequent accesses to the file.

This call will automatically position the current record pointer to the first record in order of the primary index. If another ordering is desired, the *isstart*(ISAM) call can be used to select another index.

The *filename* parameter must contain a null-terminated string, which is the name of the file to be processed.

The *mode* parameter determines the locking information. The user has three options: manual, automatic, or exclusive. Selecting the manual option indicates that the user wishes to be responsible for locking records at the appropriate times. Selecting automatic locking indicates that the user wishes to lock each record as it is read and unlock it after any subsequent function calls. Selection of exclusive locking will deny file access to anyone other than this process. More information about locking can be found in Chapter 5, "Locking". The *mode* parameter also specifies whether the file is to be opened for read, write, or read/write access.

The mode is specified by using the define macros that are found in the header file `<isam.h>`, for which a complete listing can be found in Chapter 8, "The isam.h Header File". Modes that are used in the *isopen* command are:

One of these,	arithmetically added to one of these:
ISEXCLLOCK	ISINPUT
ISMANULOCK	ISOUTPUT
ISAUTOLOCK	ISINOUT

NAME

isread — read records

SYNTAX

```
isread(isfd, record, mode)
int isfd;
char *record;
int mode;
```

DESCRIPTION

Isread is used to read records sequentially or randomly as indicated by the *mode* parameter.

When sequential processing is desired, *mode* must specify which record is to be read. It may take one of the following values:

ISCURR	current
ISFIRST	first record
ISLAST	last record
ISNEXT	next record
ISPREV	previous record

When random selection is desired, *mode* must specify the value of the record to be returned relative to the specified search value. This value may be one of:

ISEQUAL	equal to
ISGREAT	greater than
ISGTEQ	greater than or equal to

The search value is placed in the *record* buffer in the correct byte positions.

Isread will fill in the *record* with the results of the search. The *mode* is specified by using the define macros that are found in the header file `<isam.h>`. Refer to Chapter 8, "The isam.h Header File" for the contents of this file.

Isread can also read records specified by a previously set *isrecnum*. First, call *isstart*(ISAM) with *k-nparts*=0 so that the file is set to read in physical order. Then call *isread* with *mode*=ISEQUAL. This will cause *isread* to look at *isrecnum* for the desired record.

Following the successful execution of this call, the current record position and *isrecnum* will both be set to indicate the record just read.

If manual locking was specified when the file was opened and the record is to be locked before being read, the ISLOCK flag may be arithmetically added to one of the above macros. The record will then remain locked until unlocked with the *isrelease*(ISAM) call. Entire files may be locked and unlocked by using the *islock*(ISAM) and *isunlock*(ISAM) calls.

Modes that are used in the *isread* call are:

One of these,	optionally arithmetically added to
ISCURR	ISLOCK
ISNEXT	
ISFIRST	
ISGREAT	
ISEQUAL	
ISPREV	
ISLAST	
ISGTEQ	

NAME

isrelease — unlock records

SYNTAX

```
isrelease(isfd)
int isfd;
```

DESCRIPTION

Isrelease unlocks records that have been locked using the ISMANULOCK mode in the *isread*(ISAM) call. All locked records in the file indicated by *isfd* will be unlocked. More information, including examples of how to use *isrelease*, can be found in Chapter 5, "Locking".

NAME

isrename — rename an ISAM file

SYNTAX

```
isrename(oldname, newname)
char *oldname;
char *newname;
```

DESCRIPTION

Isrename will rename the file specified by the *oldname* parameter to the name specified by the *newname* parameter.

NAME

isrewcurr — rewrite current record

SYNTAX

```
isrewcurr(isfd, record)
int isfd;
char *record;
```

DESCRIPTION

Isrewcurr is used to rewrite the current record of the file indicated by *isfd* with the contents of the character array *record*. The appropriate values will be rewritten to each index that is defined. The primary key value may be changed. *Isrewcurr* is useful when the primary key is not unique and the record cannot be located and rewritten in one call. *Isrecnum* is set to indicate the current record, whose position is left unchanged.

NAME

isrewrec — rewrite record specified by record number

SYNTAX

```
isrewrec(isfd, recnum, record)
int isfd;
long recnum;
char *record;
```

DESCRIPTION

Isrewrec is used to rewrite the record specified by *recnum* in the file indicated by *isfd* with the contents of the character array *record*. *Recnum* must be a previously obtained *isrecnum* value. Each index will be appropriately updated. This call will set *isrecnum* to the value of *recnum*, while the current record position will remain unchanged.

NAME

isrewrite — rewrite record specified by primary key

SYNTAX

```
isrewrite(isfd, record)
int isfd;
char *record;
```

DESCRIPTION

Isrewrite is used to change one or more values for a record that is already in the file identified by *isfd*. The primary key is used to determine which record should be changed, and the *record* parameter contains the changes. The primary key value must be unique and may not be changed. The whole record is written to the data file. Only the changed index values will be rewritten to each index that is defined.

This is consistent with COBOL requirements for maintaining the order of records in duplicate chains. *Isrewrite* does not change the position of the current record pointer, while *isrecnum* is set to indicate this record.

NAME

isstart — select an index

SYNTAX

```
isstart(isfd, keydesc, length, record, mode)
int isfd;
struct keydesc *keydesc;
int length;
char *record;
int mode;
```

DESCRIPTION

Isstart selects the index to be used in subsequent operations. The key value to be sought should be placed in the *record* parameter, in the positions described by the *keydesc* parameter. The *keydesc* structure must describe an index that has been added previously using the *isaddindex*(ISAM) call.

The *length* parameter is used to specify the part of the key to be considered significant when doing the search. A zero indicates that the whole key is significant; a positive value is used to indicate a shorter length. If *length* is greater than zero, the response during searches will be as if the index were originally defined to have that shorter length.

The *mode* parameter may be ISFIRST, ISLAST, ISEQUAL, ISGREAT, or ISGTEQ. It is used to position the user in the file in association with the index selected by the *keydesc* argument.

ISFIRST positions the user's program in the file just before the first record in the ordering of the index specified in the *keydesc* parameter. A subsequent call to *isread*(ISAM) using the ISNEXT *mode* parameter reads the first record in the current ordering.

ISLAST positions the user's program just after the last record in that ordering. A subsequent call to *isread*(ISAM) using the ISPREV *mode* parameter reads the last record in the current ordering.

Note that if *mode* is ISFIRST or ISLAST, the parameters *length* and *record* are not needed and are not used by the *isstart* call.

Use of the ISEQUAL, ISGREAT, or ISGTEQ modes is different from the use of the ISFIRST or ISLAST modes. When using the former modes, the user's program must place the key value to be searched for in the record buffer before calling *isstart*. The value to be searched for must be placed in the location in the record buffer where the *keydesc* parameter claims the index exists.

ISEQUAL will give one of two possible results. It will either find a record whose key value is equal to that found in the appropriate positions of the record buffer parameter, or it will return an error code (-1) and set *iserrno* to ENOREC. The error code ENOREC indicates that no record with the key value specified in the *record* buffer parameter exists in the file.

ISGREAT will also give one of two responses. It will either find the next higher record whose key value is greater than that found in the *record* buffer parameter, or *isstart* will return an error condition (-1) and set *iserrno* to ENOREC.

The ISGTEQ mode parameter finds the record that has the next higher key value greater than or equal to the key value specified in the appropriate positions of the *record* buffer parameter. If no such record is found, *isstart* returns an error code (-1) and sets *iserrno* to ENOREC.

The above macros, ISFIRST, ISLAST, ISEQUAL, ISGREAT, and ISGTEQ, are defined in the header file <isam.h>.

Isstart can also be used for sequential access in physical order by specifying a previously defined key that has zero parts; that is, give a value to *keydesc* to designate a structure in which *k-nparts=0*. (see *isread*(ISAM)).

Isstart performs two basic functions. It selects the index that is to be used for subsequent reads, and it finds (but does not read) a record in the file. *Isstart* need not be used to find each record before it is read using *isread*(ISAM).

Following the successful execution of this call, the current record position and *isrecnum* will both be set to indicate this record.

NAME

isunlock — unlock an ISAM file

SYNTAX

```
isunlock(isfd)
int isfd;
```

DESCRIPTION

Isunlock is used to release an existing file-level lock for the file specified by the file descriptor *isfd*. Further discussion of locking can be found in Chapter 5, "Locking".

NAME

iswrcurr — write record and set current position

SYNTAX

```
iswrcurr(isfd, record)
int isfd;
char *record;
```

DESCRIPTION

iswrcurr writes the record passed to it in the *record* parameter to the data file identified by *isfd*. The appropriate values will be inserted into each index that is defined.

Following the successful execution of this call, the current record position and *isrecnum* will both be set to indicate this record.

NAME

iswrite — write record

SYNTAX

```
iswrite(isfd, record)
int isfd;
char *record;
```

DESCRIPTION


Iswrite writes the record passed to it in the *record* parameter to the file. The appropriate values will be inserted into each index that is defined.

Iswrite does not change the position of the current record pointer, but *isrecnum* is set to indicate this record.

X/O/P/E/N/

PORTABILITY GUIDE

RELATIONAL DATABASE LANGUAGE
(SQL)



Contents

Chapter	1	INTRODUCTION
	1.1	THE DEVELOPMENT OF DATA MANAGEMENT SYSTEMS
	1.1.1	The Importance of Data Management
	1.1.2	Linked Files
	1.1.3	Data Independence
	1.1.4	Network/Hierarchical Databases
	1.1.5	Relational Database
	1.2	THE X/OPEN DEFINITION FOR RELATIONAL DATABASE
Chapter	2	CONCEPTS
	2.1	INTRODUCTION
	2.2	SETS
	2.3	DATA TYPES AND VALUES
	2.3.1	Character strings
	2.3.2	Numbers
	2.4	COLUMNS
	2.5	TABLES
	2.6	INTEGRITY CONSTRAINTS
	2.7	THE DATABASE
	2.8	CURSORS
	2.9	EXECUTABLE SQL STATEMENTS
	2.10	EMBEDDED CONSTRUCTS
	2.11	PRIVILEGES
	2.12	TRANSACTIONS
Chapter	3	COMMON ELEMENTS
	3.1	NOTATION AND LANGUAGE STRUCTURE
	3.1.1	Notation
	3.1.2	Language Structure
	3.2	VALUES, DATA TYPES AND LITERALS
	3.2.1	Non-null Values
	3.2.2	Null Values
	3.2.3	Data Types

	3.2.4	Literals
	3.2.5	Assignments and Comparisons
	3.3	EXPRESSIONS
	3.3.1	Set Functions
	3.4	SEARCH-CONDITION
	3.4.1	Comparison Predicate
	3.4.2	BETWEEN Predicate
	3.4.3	Quantified Predicate
	3.4.4	IN Predicate
	3.4.5	LIKE Predicate
	3.4.6	NULL Predicate
	3.4.7	EXISTS Predicate
	3.5	QUERY-SPECIFICATIONS AND SUB-QUERIES
	3.5.1	Query-specifications
	3.5.2	Sub-queries
	3.5.3	Correlation-names and Correlation
Chapter	4	EMBEDDED ASPECTS
	4.1	EMBEDDED SQL HOST PROGRAM
	4.1.1	Embedded SQL Constructs
	4.1.2	Embedded Host Variables and Indicator Variables
	4.2	CURSORS
	4.3	ERROR TREATMENT
	4.4	SQL DECLARE SECTION
	4.5	THE SQL COMMUNICATION AREA (SQLCA)
	4.5.1	INCLUDE SQLCA
	4.5.2	Contents of the SQLCA
	4.6	DECLARE CURSOR
	4.7	WHENEVER STATEMENT
	4.8	SEPARATE COMPILATION
Chapter	5	EXECUTABLE SQL STATEMENTS
	5.1	DATA DEFINITION STATEMENTS
	5.1.1	ALTER TABLE
	5.1.2	CREATE INDEX
	5.1.3	CREATE TABLE
	5.1.4	CREATE VIEW
	5.1.5	DROP INDEX
	5.1.6	DROP TABLE
	5.1.7	DROP VIEW
	5.1.8	GRANT

Contents

	5.1.9	REVOKE
	5.2	DATA MANIPULATION STATEMENTS
	5.2.1	CLOSE
	5.2.2	DELETE positioned
	5.2.3	DELETE searched
	5.2.4	FETCH
	5.2.5	INSERT
	5.2.6	OPEN
	5.2.7	SELECT
	5.2.8	UPDATE positioned
	5.2.9	UPDATE searched
	5.3	TRANSACTION CONTROL STATEMENTS
	5.3.1	COMMIT
	5.3.2	ROLLBACK
Chapter	6	IMPLEMENTATION-SPECIFIC ISSUES
	6.1	LIMITS
	6.2	RESTRICTIONS ON NAMES
	6.3	DATA DEFINITION
	6.4	ASSIGNMENTS
	6.5	ERROR TREATMENT
	6.6	DECLARE CURSOR
Appendix	A	SYNTAX SUMMARY
	A.1	COMMON ELEMENTS
	A.2	EMBEDDED ASPECTS
	A.3	SQL STATEMENTS
Appendix	B	ANS X3.135 DATABASE LANGUAGE (SQL)
	B.1	FACILITY LEVEL
	B.2	EXTENSIONS
	B.3	DISCREPANCIES
Appendix	C	FUTURE DIRECTIONS

Introduction

1.1 THE DEVELOPMENT OF DATA MANAGEMENT SYSTEMS

1.1.1 The Importance of Data Management

From the earliest days of computers, methods of storing data on magnetic media have exercised computer system designers and data management has always been seen as fundamental to the successful implementation of commercial systems.

Data handling techniques have evolved through various stages, from the simple fixed format "flat file" to the sophisticated database management systems widely available today.

The simple flat file approach is clearly inadequate to support complex commercial systems, representing realistic problems in a way which is meaningful to the users of the systems. Many different types of data management products supporting complex data structures have been developed.

1.1.2 Linked Files

The earliest attempts to go beyond the constraints imposed by fixed format "flat" files were "linked" files, where records from one file contain information which allows other file(s) to be accessed for additional information.

For example, an order record could contain a reference to an item ordered in terms of its item code. To avoid having to hold the item description on every order line for the same item, a separate file containing the item description is held, directly accessible using the item code as a key. Whenever processing an order, the program uses the item code to access the second file to obtain the file description.

The indexed sequential access method (ISAM) provides support for linked files. An X/OPEN definition for ISAM is given in "INDEXED SEQUENTIAL ACCESS METHOD (ISAM)".

1.1.3 Data Independence

A major requirement of modern database management systems is a level of independence between the application's view of data and the form in which that data is stored on disc. This then allows changes to be made to the physical structure of the database without requiring all programs accessing that database to be modified appropriately. The indexed sequential access method does not offer this level of data independence.

1.1.4 Network/Hierarchical Databases

In the late 1960s and the early 1970s, a number of data management products became available, which provide tools for the support of complex data structures, with some data independence. These products share a common characteristic. For processing efficiency, the most important relationships between data records are implemented directly at the physical level by hardware pointers which give the physical disc location of related records. Chains of such pointers allow complex relationships to be represented with only a small overhead in terms of disc utilisation and reasonable processing efficiency.

The latest versions of these products are still in widespread use, particularly on mainframe computers, where the use of physical pointers and control over the physical location of data allows extremely high levels of performance to be achieved.

However, such systems suffer from a number of disadvantages which mean that they are not ideally suited to smaller machines. Because physical pointers are used to represent relationships, maintenance operations such as adding new information, relationships or merely expanding the capacity of the database is bound to be a time-consuming business and requires very specific skills to plan and implement.

1.1.5 Relational Database

The relational model was first proposed by E F Codd in 1970 and is essentially a formalisation of the linked-ISAM approach adopted intuitively by some earlier developers.

The relational model is a LOGICAL definition of the database and does not itself constrain the PHYSICAL means of implementation. It is inherently simple, based upon the concept of two dimensional tables. A number of basic operations are defined which allow tables to be combined, such as the combining of orders information and item descriptions in the example above. In each case, the result is a further table (a "derived" table) which has exactly the same characteristics as the original table in terms of the operations which can be performed upon it.

The key to the success of relational database systems, particularly on smaller systems, lies with the ease with which such systems can be implemented and maintained.

The concept of a table is inherently simple and easy to understand. When it comes to maintenance, the fact that a relational database comprises a whole series of simple tables (files) means that expanding a file or adding further fields is simple and very fast.

1.2 THE X/OPEN DEFINITION FOR RELATIONAL DATABASE

To reflect the growing significance of Relational Database systems, the X/OPEN Group has defined application interfaces embedded within high level "host" languages to a relational database management system for a free-standing database.

The widely accepted standard for such access to relational database systems is that defined in the American National Standard document Database Language SQL "ANS X3.135-1986".

The X/OPEN definition is based closely on the ANS standard but taking careful account of the capabilities of the leading relational database management systems available at the time of publication. The X/OPEN group has worked closely with the vendors of these products throughout.

Where this definition refers to current implementations or products, this refers to products which were widely available in System V environments at the time of publication. Where known, short term committed enhancements of these products have been taken into account.

The ANS SQL standard allows for two levels of compliance, Level 1 and Level 2. Most existing products comply only at Level 1 although it is expected that a significant number will have achieved full compliance with Level 2 before the end of 1987. ANS Level 1 SQL is not an adequate definition for application developers, since it leaves many areas as "implementor-defined". In preparing its definition, the X/OPEN group has examined these areas carefully and an agreed X/OPEN approach defined. Where areas remain "implementor-defined", developers should make no assumptions of behaviour based on that of any particular implementation.

Chapter 2 provides a definition of the concepts of relational database systems, an essential basis for the rest of the specification.

Chapter 3 introduces the syntax and semantics of the common elements of SQL and defines the notation used in the formal SQL syntax sections.

Chapter 4 describes how SQL constructs are embedded within the source of a high-level "host" language, making particular reference to the COBOL and C languages.

Chapter 5 contains a full and precise definition of the syntax and semantics of executable SQL statements.

Chapter 6 addresses issues which are implementation-specific and, in particular, limits which may be imposed.

Appendix A contains a summary of the SQL syntax.

Appendix B provides a detailed comparison between the X/OPEN SQL definition and the ANS X3.135-1986 standard

Appendix C identifies possible future developments of the X/OPEN SQL definition.

Concepts

2.1 INTRODUCTION

This chapter provides a definition of the concepts of relational database systems and terminology used. It provides an essential basis for the rest of the X/OPEN definition.

The chapter is structured to define the lowest level components first so that the terminology may later be used in higher level component definitions.

2.2 SETS

- a) A set is an unordered collection of distinct objects.
- b) A multi-set is an unordered collection of objects that are not necessarily distinct.
- c) A sequence is an ordered collection of objects that are not necessarily distinct.
- d) The cardinality of a collection is the number of objects in that collection. Unless specified otherwise, any collection may be empty.

2.3 DATA TYPES AND VALUES

- a) A single value has no logical subdivision and is either a null value or a non-null value.
- b) A null value is distinct from all non-null values. It represents an "unknown" or "not applicable" value.
- c) The physical representation of values in the database is implementor-defined. The logical representation of a non-null value is its display form (a literal).
- d) There is no logical representation of a null value. If a null update value is to be specified in an UPDATE statement or a null insert value is to be specified in an INSERT statement, the keyword NULL is used.
- e) There are two classes of non-null values: character string and numeric. Values belonging to different classes are not comparable. Values belonging to the same class are comparable except that character string values may not be comparable if they are taken from different character sets.
- f) A data type is a set of non-null values belonging to the same class together with a null value.

2.3.1 Character strings

- a) A character string consists of a sequence of characters taken from the system's character set.
- b) Character string values of all lengths are comparable.
- c) For every character set there is a blank character.
- d) If an implementation supports more than one character set, the means of selecting the character set and the comparability of strings over different character sets are implementor-defined. Details of character sets supported on X/OPEN systems may be found in "XVS INTERNATIONALISATION".

2.3.2 Numbers

- a) A number is either an exact numeric value or an approximate numeric value.
- b) An exact numeric value has a precision and a scale. The precision is a positive integer that determines the total number of significant decimal digits. The scale is a non-negative integer that determines the number of decimal digits to the right of the decimal point.
- c) An approximate numeric value consists of a mantissa and an exponent. An approximate numeric value has a precision which is a positive integer that determines the number of significant decimal digits of the mantissa.

2.4 COLUMNS

- a) A column is a multi-set of values that may vary over time. All values of the same column are of the same data type and are values in the same table. A value of a column is the smallest unit of data that can be selected from a table and the smallest unit of data that can be updated.
- b) A column has a specification and an ordinal position within a table. The specification of a column includes its data type and an indication of whether the column is constrained to contain only non-null values.
- c) A named column is a column of a named table or a column that inherits the specification of a named column. The specification of a named column includes its name.

2.5 TABLES

- a) A table is a multi-set of rows. A row is a non-empty sequence of values. Every row of the same table has the same cardinality and contains a value of every column of that table. The i th value in every row of a table is a value of the i th column of that table. The row is the smallest unit of data that can be inserted into a table and deleted from a table.
- b) The degree of a table is the number of columns of that table. At any time, the degree of a table is the same as the cardinality of each of its rows and the cardinality of the table is the same as the cardinality of each of its columns.
- c) A table has a specification. The specification includes a specification of each of its columns.
- d) A base table is a named table defined by a CREATE TABLE statement. The specification of a base table includes its name.
- e) A derived table is a table derived directly or indirectly from one or more other tables by the evaluation of a query-specification.
- f) A viewed table is a named derived table defined by a CREATE VIEW statement. The specification of a viewed table includes its name.
- g) Throughout this document table generally refers to viewed tables as well as base tables.
- h) A table is updatable or read-only. The INSERT, UPDATE and DELETE statements are only permitted for updatable tables.
- i) A grouped table is a set of groups derived during the evaluation of a GROUP BY clause. A group is a multi-set of rows in which all values of the grouping column(s) are equal. A grouped table may be considered as a collection of tables. Set functions may operate on the individual tables within the grouped table.
- j) A grouped view is a viewed table derived from a grouped table.

2.6 INTEGRITY CONSTRAINTS

- a) Integrity constraints define the valid states of the database by constraining the values in the base tables. Constraints may be specified to prevent two rows in a table from having the same value in a specified column or columns by using the UNIQUE option in the CREATE INDEX statement or to prevent a column from containing null values by specifying NOT NULL in the CREATE TABLE statement.
- b) Integrity constraints are effectively checked on the state of the database resulting from the execution of each SQL statement. If the base table associated with an integrity constraint does not satisfy that integrity constraint, then the SQL statement has no effect and an error is indicated.

2.7 THE DATABASE

- a) The **database** consists of the data contents and the schema contents.
- b) The **schema contents** are the definitions of all base tables, viewed tables, indexes, privileges, and user names which have been defined and which have not subsequently been explicitly or implicitly dropped.
- c) The **data contents** are all rows contained in the base tables which are currently defined by the schema contents.

2.8 CURSORS

- a) A cursor is a means for a host program to access a derived table, one row at a time.
- b) A cursor is defined by a DECLARE CURSOR statement. The specification of a cursor includes a name, a derived table and optionally an ordering of the derived rows.

2.9 EXECUTABLE SQL STATEMENTS

- a) An executable SQL statement is either a data definition statement, a data manipulation statement or a transaction control statement.
- b) A data definition statement modifies the schema contents of the database. A CREATE TABLE statement creates a base table. A CREATE VIEW statement creates a viewed table. A CREATE INDEX statement creates a base table index. A DROP TABLE statement destroys a base table. A DROP VIEW statement destroys a viewed table. A DROP INDEX statement destroys a base table index. An ALTER TABLE statement adds one or more columns to the description of a base table. A GRANT statement grants privileges on base tables and viewed tables. A REVOKE statement revokes privileges on base tables and viewed tables.
- c) A data manipulation statement operates on the data contents of the database or controls the state of a cursor. An OPEN statement opens a cursor. A CLOSE statement closes a cursor. A SELECT or FETCH statement fetches values from a table. An INSERT statement inserts rows into a table. An UPDATE statement updates the values in rows of a table. A DELETE statement deletes rows of a table.
- d) A transaction control statement terminates an active transaction. A COMMIT statement commits the database changes made by the terminated transaction. A ROLLBACK statement backs out the database changes made by the terminated transaction.

2.10 EMBEDDED CONSTRUCTS

- a) An embedded SQL host program is an application program that consists of programming language constructs and SQL constructs.
- b) The following SQL constructs may be embedded in a host program:
 - i) An SQL declare section, which defines those host program variables that are used in embedded SQL statements.
 - ii) An INCLUDE SQLCA request, which causes the SQL communication area definitions to be included in the host program. This area stores information about the last SQL statement executed.
 - iii) A DECLARE CURSOR statement, which defines a cursor.
 - iv) A WHENEVER statement, which specifies the host program action to be taken when an exception condition results during the execution of an embedded executable SQL statement.
 - v) An executable SQL statement.
- c) An embedded host variable is a normal host language variable that is referenced in an embedded SQL statement and is used to either specify a value or identify a host variable to which a column value is to be assigned.
- d) An indicator variable is an integer host language variable that is appended to a host variable in an embedded SQL statement. Its primary purpose is to indicate whether the value that the embedded host variable assumes or supplies is a null value.
- e) Embedded host variables and indicator variables must be declared in the SQL declare section.

2.11 PRIVILEGES

- a) A privilege authorises a user to perform a given category of action on a specified table. The actions that can be specified are INSERT, DELETE, SELECT and UPDATE.
- b) At any time there is a current user. It is not necessary to identify the current user by using some explicit construct in the embedded host program.
- c) The current user will become the owner of any base table, viewed table or index that is created.
- d) The owner of a table or index has all privileges on it.
- e) An SQL statement can only be executed if the current user has the appropriate privilege(s), otherwise an error is indicated.

2.12 TRANSACTIONS

- a) A transaction is a sequence of executable SQL statements that is atomic with respect to recovery and concurrency. A transaction is initiated when an SQL statement is executed and no transaction is currently active. A transaction terminates with a COMMIT or a ROLLBACK statement. If a transaction terminates with a COMMIT statement, then all changes made to the database by that transaction are made accessible to all concurrent users. If a transaction terminates with a ROLLBACK statement, then all changes made to the database are cancelled. Changes made to the database by a transaction can be perceived by that transaction, but cannot be perceived by other transactions until that transaction terminates with a COMMIT statement.
- b) If the execution of an embedded SQL host program ends while a transaction is still active, it is implementor-defined under which conditions a COMMIT or ROLLBACK statement is implicitly performed.
- c) The changes to the database caused by concurrent transactions are guaranteed to be equivalent to those caused by some serial execution of the same transactions. A serial execution is one in which each transaction executes to completion before the next transaction begins.
- d) A transaction cannot change (update or delete) a row in any table if that row is the current row of an open cursor of some concurrent transaction. (This is commonly known as "cursor stability".) An implementation may provide a higher level of isolation between concurrent transactions.
- e) The execution of an SQL statement within a transaction has no effect on the database other than the effect stated in the description for that SQL statement.

Note: Several current products do not implement the semantics of transactions, as defined above, for manipulations of the schema contents. This is described in more detail in the chapter **Implementation-Specific Issues**.

Common Elements

3.1 NOTATION AND LANGUAGE STRUCTURE

3.1.1 Notation

The following notation is used to define the SQL syntax:

- a) Words shown in upper case are keywords and must be used as specified.
- b) Words in italicised lower case are generic terms representing syntactical constructs, literals, host-identifiers or user-defined names. Where generic terms are repeated in a format, numbers may be appended to the terms to identify the separate occurrences for explanation or discussion.
- c) When square brackets, [], enclose a portion of a format, that portion is optional.
- d) When braces, { }, enclose a portion of a format, one of the options within the braces must be chosen. When only one possibility is shown, the purpose of the braces is to delimit that portion of the format to which a following ellipsis applies (see the description of an ellipsis below).
- e) A choice of options is indicated by separating the options by vertical bars.
- f) An ellipsis (...) means that the portion of the format enclosed by the immediately preceding pair of braces or brackets may be repeated.
- g) All other symbols, for example, parentheses, are mandatory and must appear as specified.

3.1.2 Language Structure

The basic unit of the language is the character. A character is any character in the system's character set. Characters are combined to form tokens and separators.

Separators

A separator is a space or a new line indication. A separator may be followed by another separator or a token. If the new line indication is a character, it cannot be used to form a token.

Tokens

A token is either a non-delimiter token or a delimiter token. A non-delimiter token must be followed by a separator or, if allowed by the syntax, a delimiter token. A non-delimiter token is a user-defined name, a host-identifier, a keyword or a numeric-literal. A delimiter token is a character-string-literal or one of the characters ",", "(", ")", "<", ">", ".", ":", "=", "*", "+", "-", and "/", or one of the character-pairs "<>", ">=", and "<=".

A user-defined name or a keyword may contain lower case letters. In these tokens, a lower case letter is equivalent to the corresponding upper case letter.

Literals are described in the section **Literals** and host-identifiers in the chapter **Embedded Aspects**; user-defined names and keywords are described below.

User-defined names

A user-defined name is a character string that must be supplied by the user to satisfy the format in which that name appears. It may consist of upper and lower case letters, digits and underscore characters except that the first character must be a letter. All the characters in a user-defined name are significant, except that case is not significant.

A user-defined name must not be the same as a keyword.

The types of user-defined names are:

Base-table-identifier
Column-identifier
Correlation-name
Cursor-name
Index-identifier
User-name
Viewed-table-identifier

The maximum length of a *user-name* is 8 characters; the maximum length of the other user-defined names is 18 characters. For certain types of user-defined names some implementations impose further restrictions, see the chapter **Implementation-Specific Issues**.

A *user-name* names a user and must be unique within the names of authorised users of the database. (The term *user-name* in this context is not the same as that associated with the System V login name.)

A *base-table-identifier* names a base table and a *viewed-table-identifier* names a viewed table. Within the base tables and viewed tables owned by a user, *base-table-identifiers* and *viewed-table-identifiers* must be unique and distinct from each other. Thus a user cannot create two tables with the same name. The term *table-identifier* is used to refer to either a *base-table-identifier* or a *viewed-table-identifier*. A *base-table-identifier*, a *viewed-table-identifier* or a *table-identifier* may be uniquely identified by being qualified by their owning *user-name*. Such optional qualification introduces further terminology as follows:

The construct [*user-name*].*base-table-identifier* is referred to as a *base-table-name*.

The construct [*user-name*].*viewed-table-identifier* is referred to as a *viewed-table-name*.

The construct [*user-name*].*table-identifier* is referred to as a *table-name*.

A *correlation-name* is an alias for a *table-name* and is specified by being paired with the *table-name* in a FROM clause.

The construct *table-name* [*correlation-name*] is referred to as a *table-reference*.

The function of a *correlation-name* is described in the section **Correlation-names and Correlation**.

A *column-identifier* names a column and must be unique within the names of the columns of the associated table. A *column-identifier* may be uniquely identified by being qualified by a *table-name* or *correlation-name*.

The construct [{*table-name* | *correlation-name*}.]*column-identifier* is referred to as a *column-name*.

An *index-identifier* names an index and must be unique within the names of indexes created by the owner of the associated base table. Some implementations impose further restrictions, see the chapter **Implementation-Specific Issues**. An *index-identifier* may be uniquely identified by being qualified by a *user-name*.

The construct [*user-name*].*index-identifier* is referred to as an *index-name*.

A *cursor-name* names a cursor and must be unique within a host program.

Although *host-identifiers* may be referenced in SQL statements, they are not user-defined names as specified in this section; they are described separately in the chapter **Embedded Aspects**.

Keywords

A keyword is a word whose presence is required when using the format in which it appears (unless it is the default). Keywords are shown in upper case in the syntax definitions.

A keyword is a reserved word in that it must not be used as a user-defined name. For reasons of compatibility, the list of keywords which are reserved in this way is extended to include all the keywords in American National Standard Database Language SQL ("X3.135-1986"). Some implementations may reserve additional words, see the chapter **Implementation-Specific Issues**.

Keywords Used in X/OPEN SQL

ADD	ALL	ALTER	AND	ANY
AS	ASC	AVG	BEGIN	BETWEEN
BY	CHAR	CLOSE	COMMIT	CONTINUE
COUNT	CREATE	CURRENT	CURSOR	DECIMAL
DECLARE	DELETE	DESC	DISTINCT	DROP
END	EXEC	EXISTS	FETCH	FLOAT
FOR	FOUND	FROM	GOTO	GRANT
GROUP	HAVING	IN	INCLUDE	INDEX
INSERT	INTEGER	INTO	IS	LIKE
MAX	MIN	NOT	NULL	OF
ON	OPEN	OR	ORDER	PUBLIC
REVOKE	ROLLBACK	SECTION	SELECT	SET
SMALLINT	SQL	SQLCA	SQLERROR	SQLWARNING
SUM	TABLE	TO	UNION	UNIQUE
UPDATE	USER	VALUES	VIEW	WHENEVER
WHERE	WORK			

Additional Keywords in "X3.135-1986" SQL

AUTHORIZATION	CHARACTER	CHECK	COBOL	DEC
DOUBLE	ESCAPE	FORTRAN	GO	INDICATOR
INT	LANGUAGE	MODULE	NUMERIC	OPTION
PASCAL	PLI	PRECISION	PRIVILEGES	PROCEDURE
REAL	SCHEMA	SOME	SQLCODE	WITH

3.2 VALUES, DATA TYPES AND LITERALS

A value is the smallest unit of data that can be manipulated in SQL and is either a null value or a non-null value.

A data type is a set of representable values. Each data type includes a null value that is distinct from all non-null values.

A literal is the display form of a non-null value and is used to logically represent such a value within an SQL statement. A null value cannot be represented by a literal.

3.2.1 Non-null Values

There are two classes of non-null values, character string and numeric.

A character string is a non-empty sequence of characters taken from some character set and has a length, which is the number of characters in the sequence.

A number is either an exact numeric value or an approximate numeric value.

An exact numeric value has a precision and scale. The precision determines the number of significant decimal digits. The scale determines the number of decimal digits to the right of the decimal point.

An approximate numeric value consists of a mantissa and an exponent. The mantissa is a signed numeric value and the exponent is a signed integer value that specifies the magnitude of the mantissa. An approximate numeric value has a precision which determines the number of significant decimal digits in the mantissa.

3.2.2 Null Values

A null value is a type-dependent special value that is distinct from all non-null values and represents a value that is unknown or not applicable.

Only columns that are not constrained to contain only non-null values may contain a null value. When inserting rows into a table, columns that do not have corresponding insert values must allow, and are assigned, null values.

The use of an *indicator-variable* in conjunction with an embedded host variable enables a *host-variable-reference* to evaluate to, or to be assigned, a null value (see the chapter **Embedded Aspects**).

Although a null value cannot be represented by a *literal*, the keyword NULL, which is always used in association with a table column, is used to represent the null value; it may only be used as indicated by the syntax.

If the value of any operand in an *expression* is the null value then the value of the *expression* is the null value. (In particular, subtracting a null value from a null value evaluates to null and not to zero.)

Null Values and Explicit (i.e. user) Comparisons

Apart from the special case of the NULL predicate, comparing a null value with any value, even another null value, always evaluates to the unknown truth value.

The NULL predicate utilises the NULL keyword in a comparison of the form IS [NOT] NULL and is the only method of determining whether or not a value is a null value. (Note that an *indicator-variable* cannot be used in conjunction with an embedded host variable that is being used to provide a comparison value.)

Null Values and Implicit (i.e. System) Comparisons

The following table summarises how null values are handled in system-generated comparisons.

Context	Treatment
GROUP BY	Null grouping column values are equal.
UNIQUE INDEX	Null values are allowed and are equal.
ORDER BY	Null values are equal and are either greater than or less than all non-null values.
SELECT DISTINCT	Null values are equal.
Set functions	Null values are eliminated regardless of whether DISTINCT is specified except that COUNT(*) includes all null values.

3.2.3 Data Types

A data type describes the source of a value in terms of the type of value it may contain or represent. Thus, the basic data types are character string, exact numeric and approximate numeric. These are referred to as generic data types to distinguish them from the named data types (see below).

The sources of values are expressions, host variables, literals, set functions and table columns. Host variables and their associated data types are described in the chapter **Embedded Aspects**; the association of data types with the other value sources is described in the sections that follow.

Data Types and Table Columns

The columns of a base table have named data types associated with them when the table is created and when columns are added to the table. A named data type is a generic data type with defined attributes (for example, length or precision). The following named data types are supported:

Data Type	Description	Range
CHAR(n)	Character string of fixed length n	$1 \leq n \leq 240$
SMALLINT	Exact numeric, signed, precision 5, scale zero	Absolute value less than 32 768
INTEGER	Exact numeric, signed, precision 10, scale zero	Absolute value less than 2,147,483,648
DECIMAL(p,s)	Exact numeric, signed, precision p, scale s.	$1 \leq p \leq 15; 0 \leq s \leq p$
FLOAT	Approximate numeric, signed, mantissa precision 15.	Zero or absolute value $1.0E-38$ to $1.0E+38$

Some implementations may support larger ranges than shown. Application developers may safely assume that all systems support the minimum values defined.

The columns of a derived table obtain their data types as follows:

The columns of the derived table defined by a *query-specification* derive their data types from the *expressions* that provide the column definitions. The columns of a viewed table inherit the data types of the corresponding result columns of the *query-specification* that defines the view. The columns of the derived table defined by a cursor specification inherit the data types of the corresponding result columns of the first *query-specification* in the cursor specification.

Thus a column of a derived table may have either a named data type or a generic data type.

Data Types and Expressions

If an *expression* contains a single operand, the data type of the *expression* is the data type of that operand.

The data type of the result of an arithmetic operation depends on the data types of the two operands and is determined as follows:

If the data type of either operand is approximate numeric, the data type of the result is approximate numeric with implementor-defined precision and range of magnitude.

If the data type of the result is approximate numeric, the result must be within the implementor-defined range of magnitude.

If the data type of both operands is exact numeric, the data type of the result is exact numeric with implementor-defined precision and range, and scale dependent on the operation as follows:-

- If the operation is addition or subtraction, the scale is the larger of the scales of the two operands.
- If the operation is multiplication, the scale is the sum of the scales of the two operands.

c) If the operation is division, the scale is implementor-defined.

If the data type of the result is exact numeric and the operator is not division, the result must be exactly representable within its data type. If the data type of the result is exact numeric and the operator is division, the result must be representable within its data type without losing any leading significant digits.

Data Types and Set Functions

The data type of the result of a *set-function-reference* is defined by the following table:

Function Name	Argument	Data type of result
COUNT	Any	Exact numeric, scale zero, implementor-defined precision and range
MAX	Any	As argument
MIN	Any	As argument
SUM	Exact numeric, scale s	Exact numeric, scale s, implementor-defined precision and range
SUM	Approximate numeric	Approximate numeric, implementor-defined precision and range of magnitude
AVG	Exact numeric	Exact numeric, implementor-defined precision, scale and range of magnitude
AVG	Approximate numeric	Approximate numeric, implementor-defined precision and range of magnitude

3.2.4 Literals

A *literal* is a sequence of characters representing a value. *Literals* are classified according to the values that they represent.

Character-string-literals

A *character-string-literal* represents a character string and consists of a sequence of characters delimited at each end by the single quote character.

That is, it has the following format:

'{character}...'

Within the delimiters, a single quote character is represented by two consecutive single quote characters. The value of the *literal* is the value of the sequence of characters within the delimiters. The data type of a *character-string-literal* is character string.

Numeric-literals

A *numeric-literal* represents a number and consists of a character string whose characters are selected from the digits 0 through 9, the plus sign, the minus sign,

the decimal point and the character "E".

A *numeric-literal* is either an *exact-numeric-literal* representing an exact numeric value or an *approximate-numeric-literal* representing an approximate numeric value.

An *exact-numeric-literal* has the following format:

$$[+|-]\{ \textit{unsigned-integer} [\textit{.unsigned-integer}] \\ | \textit{unsigned-integer} . \\ | \textit{.unsigned-integer} \}$$

where *unsigned-integer* is defined as $\{ \textit{digit} \} \dots$

The data type of an *exact-numeric-literal* is exact numeric, its precision is the number of digits that it contains and its scale is the number of digits to the right of the decimal point. The value of an *exact-numeric-literal* is derived from the normal mathematical interpretation of the specified notation.

An *approximate-numeric-literal* has the format *mantissa**E**exponent*, where the *mantissa* is an *exact-numeric-literal* and the *exponent* has the form

$$[+|-]\textit{unsigned-integer}.$$

The data type of an *approximate-numeric-literal* is approximate numeric and its precision is the precision of its *mantissa*.

The value of an *approximate-numeric-literal* is the product of the value represented by the *mantissa* with the number obtained by raising the number 10 to the power represented by the *exponent*.

USER

The keyword USER represents the name of the current user and is considered to be equivalent to a *character-string-literal* whose value is the current *user-name*. It may be specified in place of a *character-string-literal* wherever a format or description allows a *character-string-literal* to be used.

3.2.5 Assignments and Comparisons

The fundamental operations of the language are assignment and comparison. In each case, the general rule is that numbers and character strings are compatible with themselves but not with each other. Thus, numbers and character strings cannot be compared, numbers cannot be assigned to character string table columns or host variables, and character strings cannot be assigned to numeric table columns or host variables.

Comparisons

All numbers are comparable and are compared according to their algebraic value.

All character strings taken from the same character set are comparable and are compared from left to right with respect to the collating sequence of the character set they are taken from.

If two character strings of different lengths are to be compared, the shorter is conceptually padded on the right with blank characters to make it the same length as the other before the comparison is carried out.

Assignments

Assigning Null Values

1. If the null value is to be assigned to a table column, the column must be allowed to contain null values and its value is set to the null value.
2. If the null value is to be assigned to a host variable, the host variable must have an associated indicator variable and that indicator variable is set to -1. The content of the host variable is undefined.

Setting Indicator Variables

1. If the null value is assigned to a host variable, the required indicator variable is set to -1 (see 2 above).
2. If a non-null value is assigned to a host variable, the indicator variable, if present, is set to zero unless a character string has been truncated (see 2 below) when it is set to the original length of the character string. Some implementations currently do not follow this treatment, see the chapter **Implementation-Specific Issues**.

Character String Assignments

The following rules apply to assigning a non-null character string of length S (source) to a table column or host variable whose data type is character string of length D (destination). Trailing blank characters are included in the length of a source character string. Note that for C , a character string of length D requires an array of $D+1$ elements, and the last position is referred to as element $[D]$ of the array.

1. If the assignment is to a table column, S must not be larger than D .
2. If the assignment is to a host variable and S is larger than D , the leftmost D characters of the character string are assigned and SQLWARN0 and SQLWARN1 are set to W to indicate that truncation has occurred. For C , a null byte ($\backslash 0$) is placed in element $[D]$ of the array.
3. If S is smaller than D , the character string is assigned to the leftmost S character positions of the destination field. For COBOL, the remaining $D-S$ character positions are filled with the blank character. For C , a null byte ($\backslash 0$) is placed in element $[S]$ of the array.
4. If S is equal to D , the character string is assigned to the destination field. For C , a null byte ($\backslash 0$) is placed in element $[D]$ of the array.

Numeric Assignments

The following rules apply to assigning a non-null numeric value to a table column or host variable whose data type is exact numeric or approximate numeric.

1. If the data type of the destination field is exact numeric, there must be a representation of the numeric value in the data type of the destination field that does not cause the whole part of the number (that is, the leading significant digits) to be truncated, and the destination field is set to that representation. The fractional part of the number (that is, the trailing significant digits) may be truncated as necessary and it is implementor-defined whether warning flags and/or indicator variables will be set.
2. If the data type of the destination field is approximate numeric, the numeric value must be within the range of magnitude of the destination field. The destination field is set to an approximation of the numeric value that has the precision of the destination field.

3.3 EXPRESSIONS

An *expression* represents a single value. It consists of a *host-variable-reference*, a *literal*, a *column-name*, a *set-function-reference*, or any combination of these primary components connected by arithmetic operators (provided that all the operands are numeric).

The following arithmetic operators are allowed (in descending order of precedence):

- + , - plus, minus (unary)
- *, / multiplication, division
- + , - plus, minus (binary)

The order of execution of operations at the same precedence level is from left to right. Parentheses may be used to depart from the above precedence order since, when used in an *expression*, they determine the binding of *expression* components to arithmetic operators.

If the value of any primary component is the null value, the value of the *expression* is the null value.

An *expression* directly contained within the *search-condition* of a WHERE clause must not reference a column derived from a set function.

3.3.1 Set Functions

Apart from the special case of COUNT(*) (see below), set functions operate on the collection of values in one column of some table or group of a grouped table, and produce a single value (defined as follows) as their result:

Function name	Result
COUNT	Number of values in the column
SUM	Sum of the values in the column
AVG	Average of the values in the column
MAX	Largest value in the column
MIN	Smallest value in the column

The general format of a *set-function-reference* is as follows:

set-function-name ([DISTINCT] *argument*)

If DISTINCT is specified, redundant duplicate values of the *argument* are eliminated before the function is applied. If DISTINCT is not specified, duplicate values are retained. In both cases, any null *argument* values are eliminated before the function is applied. The elimination of null values is signalled by SQLWARN0 and SQLWARN2 being set to W.

COUNT(*argument*) is not supported but the special case of COUNT(*) is provided to count all rows in a table without duplicate elimination. If the *argument* happens to be an empty set, COUNT returns a value of zero, the other functions return the null value.

If DISTINCT is specified, the *argument* must be a simple *column-name* and the *set-function-reference* must stand alone, that is, it cannot be used inside an *expression* that contains binary operators. If DISTINCT is not specified, the *argument* may be an *expression* but it must contain at least one *column-name* and may not contain any *set-function-references*.

A *column-name* specified as the *argument* of a *set-function-reference* must neither reference a column derived from a set function nor be a correlated reference.

Functions AVG and SUM can be applied only to numeric columns. When calculating an average, the sum must be within the range of the data type of the result.

A *set-function-reference* may be specified only in an *expression* in a SELECT clause or in an *expression* that is directly contained within the *search-condition* of a HAVING clause.

3.4 SEARCH-CONDITION

Search-conditions are used in WHERE and HAVING clauses to qualify the selection of rows and and groups, respectively. They consist of one or more predicates combined with logical operators. Predicates are negated using the logical operator NOT and are connected using the logical operators AND and OR.

The order of precedence among the logical operators is NOT (highest), followed by AND, followed by OR. The order of evaluation at the same precedence level is from left to right.

Parentheses may be used to depart from the above precedence order since, when used in a *search-condition*, they determine the binding of predicates to logical operators.

Applying a predicate to a given row of a table or a given group of a grouped table results in a truth value of true (T), false (F) or unknown (U). NOT (true) is false, NOT (false) is true and NOT (unknown) is unknown. The truth tables for AND and OR are given below.

AND	T	F	U
T	T	F	U
F	F	F	F
U	U	F	U

OR	T	F	U
T	T	T	T
F	T	F	U
U	T	U	U

Thus, for a given row or group, a *search-condition* evaluates to true, false or unknown. Qualifying rows or groups are those for which the *search-condition* evaluates to true.

A *column-name* or *expression* specified in a *search-condition* is directly contained in that *search-condition* if the *column-name* or *expression* is not specified within a *set-function-reference* or a *sub-query* of that *search-condition*.

Whenever two operands are implicitly or explicitly compared in a predicate, their data types must be comparable. When a host variable is supplying a comparison value for use in a predicate, an indicator variable must not be present.

The various types of predicate are described in the following sections.

3.4.1 Comparison Predicate

A comparison predicate compares two values and has the form:

expression-1 *comparison-operator* { *expression-2* | (*sub-query*) }

where *comparison-operator* may be any of the following:

=	equal to
<>	not equal to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to

Within the context of a comparison predicate, a *sub-query* must result in either a single value or an empty set. If the result of the *sub-query* is an empty set, the result of the predicate is unknown. If the result of the *sub-query* is a single value, the same rules apply as for *expression-2*.

If either *expression-1* or *expression-2* evaluates to the null value, the result of the predicate is unknown.

If neither *expression-1* nor *expression-2* evaluates to the null value, the result of the predicate is the result of the comparison .

3.4.2 BETWEEN Predicate

A BETWEEN predicate tests whether a value is within a range of values and has the form:

expression-1 [NOT] BETWEEN *expression-2* AND *expression-3*

If NOT is not specified, the predicate is equivalent to:

expression-1 >= *expression-2* AND *expression-1* <= *expression-3*

If NOT is specified, the result of the predicate is the result of applying the logical operator NOT to the identical predicate without NOT specified.

3.4.3 Quantified Predicate

A quantified predicate compares a value with a number of derived values and has the form:

expression *comparison-operator* {ALL | ANY} (*sub-query*)

where *comparison-operator* is as described in the section **Comparison Predicate**.

If ALL is Specified

The result of the predicate is true if the *sub-query* results in an empty set or if the comparison is true for every value returned by the *sub-query*. The result of the predicate is false if the comparison is false for at least one value returned by the *sub-query*. The result of the predicate is unknown if it is neither true nor false.

If ANY is Specified

The result of the predicate is true if the comparison is true for at least one value returned by the *sub-query*. The result of the predicate is false if the *sub-query* results in an empty set or if the comparison is false for every value returned by the

sub-query. The result of the predicate is unknown if it is neither true nor false.

3.4.4 IN Predicate

An IN predicate compares a value with a list of values or with a number of derived values and has the form:

expression [NOT] IN {(*value* { , *value* } ...) | (*sub-query*)}

where *value* is a *literal* or a *host-variable-reference*.

The operator IN is equivalent to the operator = ANY. Thus, when NOT is not specified, the result of the predicate is true if the implied equality comparison is true for at least one specified or derived comparison value and is false if the implied equality comparison is false for every specified or derived comparison value or if the *sub-query* results in an empty set; the result of the predicate is unknown if it is neither true nor false. If NOT is specified, the result of the predicate is the result of applying the logical operator NOT to the identical predicate without NOT specified.

3.4.5 LIKE Predicate

A LIKE predicate compares a character string with a pattern and has the form:

column-name [NOT] LIKE *pattern-value*,

where *pattern-value* is a *character-string-literal* or a *host-variable-reference*.

Column-name must reference a character string column and if a *host-variable-reference* is specified it must reference a character string variable.

If the value of the column referenced by *column-name* is the null value, the result of the predicate is unknown. If NOT is not specified, the result of the predicate is true or false depending on whether or not the value of the column referenced by *column-name* conforms to the specified pattern. If NOT is specified, the result of the predicate is the result of applying the logical operator NOT to the identical predicate without NOT specified.

Within the character string represented by *pattern-value*, characters are interpreted as follows:

The underscore character (*_*) stands for any single character.

The percent character (*%*) stands for any sequence of zero or more characters.

All other characters stand for themselves.

Thus, for example, LIKE '*%A%*' is true for any column value that contains the character A and LIKE '*B_ _*' is true for any column value that is three characters long and starts with the character B.

3.4.6 NULL Predicate

A NULL predicate compares a value against the null value and has the form:

column-name IS [NOT] NULL

If NOT is not specified, the result of the predicate is true if the value of the column referenced by *column-name* is the null value, otherwise the result of the predicate is false.

If NOT is specified, the result of the predicate is true if the value of the column referenced by *column-name* is not the null value, otherwise the result of the predicate is false.

Note that this is the only predicate that provides a way of testing for the presence, or absence, of null values.

3.4.7 EXISTS Predicate

An EXISTS predicate tests for the existence of a row satisfying some condition and has the form:

EXISTS (*sub-query*)

The result of the predicate is true if the *sub-query* does not result in an empty set, otherwise the result of the predicate is false.

3.5 QUERY-SPECIFICATIONS AND SUB-QUERIES

A *query-specification* is the basic construct for specifying data to be retrieved. It is the means of deriving a table by selecting specific fields from one or more tables subject to some specified condition.

A *sub-query* is a limited form of *query-specification* used to derive values for use in a predicate.

3.5.1 Query-specifications

A *query-specification* has the form:

```
SELECT [ALL | DISTINCT] { * | expression [, expression]... }  
FROM table-reference [, table-reference]...  
[WHERE search-condition]  
[GROUP BY column-name [, column-name]... ]  
[HAVING search-condition]
```

A *query-specification* is best explained by considering the effects of the various clauses in the order in which they are conceptually applied.

- Step 1. A Cartesian product is formed from the tables referenced by the FROM clause.
- Step 2. The result of Step 1 is reduced by the elimination of all rows that do not satisfy the *search-condition* specified by the WHERE clause.
- Step 3. The result of Step 2 is grouped by the values of the columns identified by the GROUP BY clause. That is, the result is arranged into groups such that within any one group all rows have the same values for the grouping columns. If a grouping column contains any null values, these values are considered to be equal and give rise to only a single group.
- Step 4. The result of Step 3 is reduced by the elimination of all groups that do not satisfy the *search-condition* specified by the HAVING clause. *Expressions* in the HAVING clause must be single-valued per group. Thus each *column-name* directly contained within the *search-condition* must reference a grouping column or be a correlated reference, and if a *sub-query* contained in the *search-condition* contains a correlated reference that correlated reference must be to a grouping column.

If the GROUP BY clause is not present, the result of Step 2 is treated as a single group with no grouping columns.

- Step 5. Input to this stage is either a grouped table from Step 4 or a non-grouped table from Step 2. The next step is to generate result rows based on the result columns specified by the SELECT clause.

If the SELECT clause contains a sequence of *expressions*, each *expression* represents a result column. If the SELECT clause contains the asterisk (*)

character, the result columns are all the columns of the tables referenced by the FROM clause.

If the input is a grouped table, each group generates one result row and the result columns must be single-valued per group. Therefore, any column referenced by the result column specification must either be a grouping column or be referenced within the *argument* of a *set-function-reference*.

If the input is not a grouped table, each row generates one result row unless a result column is derived from a set function, in which case the *query-specification* results in only a single row and any column referenced by the result column specification must be referenced within the *argument* of a *set-function-reference*.

- Step 6. The result of Step 5 is reduced by the elimination of duplicate rows if DISTINCT is specified in the SELECT clause.

Thus, if ALL is specified or DISTINCT is not specified, the result of the *query-specification* is the table resulting from Step 5, whereas, if DISTINCT is specified, the result of the *query-specification* is the table derived from the result of Step 5 by eliminating any redundant duplicate rows.

Excluding *sub-queries*, the keyword DISTINCT must not be used more than once within a *query-specification*. For example, if DISTINCT is used to eliminate duplicate result rows, it cannot also be used to eliminate duplicates from the *argument* of a set function.

Deriving from a Grouped View

If the FROM clause references a grouped view, it must contain exactly one *table-reference*, the *query-specification* must not contain a WHERE, GROUP BY or HAVING clause and the result column specification must not contain a *set-function-reference*.

Updatability

A table is either updatable or read-only. The operations of insert, update and delete are permitted for updatable tables and are not permitted for read-only tables.

All base tables are updatable. A viewed table is updatable if it is derived from an updatable *query-specification*. The result table of a cursor is updatable if neither ORDER BY nor UNION is specified in the cursor specification and the result table is derived from an updatable *query-specification*.

A *query-specification* is updatable if and only if the following conditions are satisfied:

- a) The FROM clause contains a single *table-reference* and that *table-reference* refers to an updatable table.

- b) Neither the GROUP BY clause nor the HAVING clause is present.
- c) DISTINCT is not specified.
- d) The WHERE clause does not include a *sub-query*.
- e) All the result columns are derived from *column-names*. That is, the result columns are specified using either the asterisk notation or a sequence of *expressions* each one of which consists of a single *column-name*.

3.5.2 Sub-queries

A *sub-query* has the form:

```
SELECT [ALL | DISTINCT] { * | expression }  
      FROM table-reference [, table-reference]...  
      [WHERE search-condition]  
      [GROUP BY column-name [, column-name]... ]  
      [HAVING search-condition]
```

A *sub-query* is a limited form of *query-specification* used to provide a multi-set of values or a multi-set of rows within a predicate. *Sub-queries* must be enclosed in parentheses and may be nested.

A *sub-query* can be used only on the right hand side of a comparison predicate, quantified predicate or IN predicate or as the subject of an EXISTS predicate. The result of the *sub-query* is substituted into the predicate of the outer query.

When a *sub-query* is used in a comparison, quantified or IN predicate it is used to derive a multi-set of values and can therefore only have a single result column.

When a *sub-query* is used in an EXISTS predicate it is used to derive a multi-set of rows, but it is the existence of one or more result rows rather than the values in these rows that is significant; therefore, in this case, the result columns would normally be specified using the shorthand asterisk notation although a single result column may be explicitly specified.

Since the right hand side of a comparison predicate requires only a single value, a *sub-query* used in this position must result in at most a single value. It follows that, in this case, there must be no grouping involved in the derivation and as a result the *sub-query* must not contain a GROUP BY or HAVING clause and the FROM clause must not reference a grouped view.

The rules relating to deriving from a grouped view are the same as those for a *query-specification* and the same restriction applies regarding multiple use of the keyword DISTINCT.

3.5.3 Correlation-names and Correlation

As noted in earlier sections, a *correlation-name* is an alias for a *table-name* and is specified by being paired with the *table-name* in a FROM clause. Thus a *correlation-name* can be used in lieu of a *table-name* when qualifying *column-*

names. This aspect and qualification in general is addressed below.

Clearly an alias can be used as a shorthand version of a *table-name* but its more important application is as a means of distinguishing or relating table references. These aspects are also addressed below.

Scope of Table-names

If a *table-reference* in a FROM clause does not contain a *correlation-name*, the *table-name* is the name that denotes that occurrence of the table and, as such, may be used to qualify *column-identifiers* within its scope. If a *table-reference* in a FROM clause contains a *correlation-name*, the *correlation-name* is the name that denotes that occurrence of the table and, as such, may be used to qualify *column-identifiers* within its scope. The scope of a name that denotes a table occurrence is the entire innermost *sub-query*, *query-specification* or SELECT statement that contains the FROM clause.

Within a FROM clause, *table-names* that denote table occurrences must be unique and *correlation-names* must be unique and distinct from the *table-identifiers* of *table-names* that denote table occurrences.

The scope of a *table-name* in an UPDATE or searched DELETE statement is the entire statement.

If a *column-name* contains a qualifier, the *column-name* must appear within the scope of that qualifier. If the *column-name* is within the scope of more than one qualifier with the specified name, the one with the most local scope is assumed. The table associated with the qualifier must contain a column with the specified *column-identifier*. If a *column-name* does not include a qualifier, the *column-name* must be within the scope of, and is assumed to be qualified by, a qualifier whose associated table contains a column with the specified *column-identifier*. If there is more than one possible qualifier, there must be only one with the most local scope and that one is assumed.

Multiple Table-references Associated with the Same Table

The two situations where this can occur are described below with respect to the general case of two *table-references*.

- a) Within the same FROM clause.

A typical example of this situation is joining a table to itself, that is, comparing two rows in the same table. In such a situation, we have two simultaneous distinct references to the same table and two distinct qualifiers are required, and must be explicitly specified, in order to distinguish between the two occurrences. Thus, at least one of the *table-references* must contain a *correlation-name*.

- b) *Sub-query* and outer query referencing the same table.

This encompasses two situations:

- i) The *sub-query* refers only to its own occurrence

In this case, we have two non-simultaneous distinct references to the same

table that do not need to be distinguished although it may be useful to provide and use distinct qualifiers in order to clarify the distinction.

- ii) The *sub-query* refers to both occurrences
In this case we again have two simultaneous distinct references to the same table and two distinct qualifiers are required to distinguish between the two occurrences. Thus, at least one of the *table-references* must contain a *correlation-name*. Note, however, that in this case only the *sub-query* references to the occurrence in the outer query need to be explicitly qualified since the scope rules resolve the other references.

Correlation

It is not always the case that a *sub-query* can be evaluated once and for all and the result substituted into the predicate of the outer query. If the value of the *sub-query* depends on the value of a column in the row of some outer query, the *sub-query* has to be evaluated once for each row of the outer query. Such a *sub-query* is termed a correlated *sub-query* and the reference to the column of the outer query is termed a correlated reference.

Correlation is exemplified by the type of *sub-query* described in b)ii) above, however, the type of *sub-query* described in b)i) above is not of the correlated variety since the table occurrences are different.

A correlated reference need not be explicitly qualified unless this is required for uniqueness of reference, however, it is a useful practice to associate a *correlation-name* with the relevant *table-name* in the FROM clause of the outer query and to qualify the correlated reference with this name in order to make the correlation clear.

Embedded Aspects

4.1 EMBEDDED SQL HOST PROGRAM

An embedded SQL host program is a database application program that consists of programming language statements and embedded SQL constructs. X/OPEN defines the way in which SQL statements can be hosted in COBOL and C. Many products support further languages.

The SQL constructs that may be embedded in a host program fall into two categories, declaratives and statements. SQL declaratives are specified in the data declaration part of a program and provide a means of communicating data values and information between the host program and the database system. SQL statements are specified in the procedural part of a program and provide a means of accessing and manipulating the contents of the database.

The SQL declaratives are:

- i) the SQL declare section,
- ii) the INCLUDE SQLCA declarative.

The SQL statements are:

- i) the DECLARE CURSOR statement,
- ii) the WHENEVER statement,
- iii) the executable SQL statements defined in the chapter **Executable SQL Statements**.

Host programs may include the data definition statements described in the section **Data Definition Statements**, however, for correct processing of the data manipulation statements, it is required that the schema contents of the database referenced by these statements are available at the time of (pre-)compilation. In particular, the CREATE TABLE, ALTER TABLE and CREATE VIEW statements for all referenced tables must be executed prior to (pre-)compilation.

For convenience, we shall dispense with the adjective "embedded" for the rest of this chapter and refer simply, without loss of clarity, to SQL constructs, declaratives and statements.

4.1.1 Embedded SQL Constructs

This section covers the coding requirements for SQL constructs. Note that although an SQL declare section is a single construct, its header (BEGIN DECLARE SECTION) and trailer (END DECLARE SECTION) are considered as individual constructs in the context of this section.

SQL constructs are distinguished from programming language statements by being delimited by the special prefix EXEC SQL and a host language dependent special

terminator.

The prefix EXEC SQL, the SQL declare section header and the SQL declare section trailer must each be specified within one line.

An SQL construct must not contain any comments. SQL constructs embedded in host language comments are treated as comments.

Lower case letters are allowed for SQL keywords and user-defined names.

Special Rules for COBOL

The SQL terminator is END-EXEC.

SQL constructs must not be contained within library text processed by a COPY statement.

An SQL construct must be the only construct on a line (that is Area A and Area B), however, it may be followed by any appropriate COBOL punctuation character.

SQL constructs must be specified in area B. The only token that may be split across lines is a character string literal and this is achieved using the COBOL continuation facility.

The placing of the SQL declaratives is addressed by the sections **SQL Declare Section** and **SQL Communication Area (SQLCA)**.

SQL statements must be specified in the program's Procedure Division and may appear anywhere an imperative-statement is allowed.

Special Rules for C

The SQL terminator is a semicolon (;).

SQL constructs must not be contained within an include file.

An SQL construct must be the only construct on a line.

A token must not be split across lines.

The placing of the SQL declaratives is addressed by the sections **SQL Declare Section** and **SQL Communication Area (SQLCA)**.

SQL statements may be specified anywhere a C statement may be specified within a function block.

4.1.2 Embedded Host Variables and Indicator Variables

Embedded host variables are normal host language variables that are used in SQL statements to either specify values or communicate database values to the host program.

Additional information about the value supplied or assumed by an embedded host variable may be provided by combining the embedded host variable with an indicator variable. An indicator variable is a two-byte integer host language variable.

Indicator variables are used as follows:

1. To indicate whether the value assigned to the associated embedded host variable by an executable SQL statement is the null value. A value of -1 indicates the null value.
2. To indicate whether the value supplied by the associated embedded host variable to an executable SQL statement is the null value. A value of -1 indicates the null value.
3. To indicate whether a character string assigned to the associated embedded host variable by an executable SQL statement has been truncated. A positive value indicates truncation has occurred.

When an embedded host variable or an indicator variable is specified in an SQL statement, it must be immediately preceded by a colon (:). An indicator variable, when present, is specified immediately after its associated embedded host variable. Thus, when both variables are present, a reference to a host variable would appear as *:host-identifier:host-identifier*.

We use the term *host-variable-reference* to refer to an embedded host variable with an optionally appended indicator variable. Thus, syntactically, a *host-variable-reference* is defined as the construct

embedded-variable-name[*indicator-variable*]

where each of *embedded-variable-name* and *indicator-variable* equates to a *host-identifier* preceded by a colon.

Embedded host variables and indicator variables must be declared in the SQL declare section. The host identifiers representing these variables may be any valid host variable name and may be the same as an SQL keyword or user-defined name. Some implementations currently impose restrictions on the names of these variables, see the chapter **Implementation-Specific Issues**.

A *host-variable-reference* used in the INTO clause of a FETCH or SELECT statement identifies a host variable to which a column value is to be assigned. In all other contexts, a *host-variable-reference* identifies a host variable that is supplying a value. When the referenced host variable is supplying a comparison value for use in a predicate, an indicator variable must not be present; in all other cases, an indicator variable may be included.

Data Types and Embedded Host Variables

An embedded host variable must have a description that corresponds to one of the SQL data types, and the data type of the host variable is considered to be that corresponding data type. All embedded host variables are conceptually allowed to contain null values.

The correspondence table for COBOL is as follows:

SQL data type	COBOL equivalent
CHAR(n)	PIC X(n) ; 1 <= n <= 240
SMALLINT	PIC S9(5) or PIC S9(5) COMP-3 or PIC S9(4) COMP
INTEGER	PIC S9(10) or PIC S9(10) COMP-3 or PIC S9(9) COMP
DECIMAL(m+n,n)	PIC S9(m)V9(n) or PIC S9(m)V9(n) COMP-3 or PIC S9(m)V9(n) COMP m > 0;n >= 0;m+n <= 15 Note: V is required
FLOAT	No equivalent

The correspondence table for C is as follows:

SQL data type	C equivalent
CHAR(n)	char[n+1]; 1 <= n <= 240
SMALLINT	short or int
INTEGER	long
DECIMAL(p, s)	No equivalent
FLOAT	double

Only portable C int variables are considered; that is the absolute value for a C int variable must be less than 32 768.

4.2 CURSORS

A cursor is a means for a host program to access a table, one row at a time.

A cursor is defined by a `DECLARE CURSOR` statement. The specification of a cursor includes a name, a derived table and optionally an ordering of the derived rows.

A cursor is either in the open or closed state. A cursor in the open state designates an active set and a position relative to that active set. An active set designates a derived table and an ordering of the rows of that table. If no order was specified in the `DECLARE CURSOR` statement, the rows of the active set have an implementor-defined order.

The position of a cursor in the open state is either before a certain row, on a certain row, or after the last row. If the position is on a row, then that row is the current row of the active set. A cursor may be before the first row or after the last row even though the active set is empty.

A `FETCH` statement advances the position of the cursor to the next row of the active set and retrieves the values of the columns of that row. A positioned `UPDATE` statement updates the current row of the active set. A positioned `DELETE` statement deletes the current row of the active set.

If the position of the cursor is before a row and a new row is inserted at that position, then the effect, if any, on the position of the cursor is implementor-defined.

If the position of the cursor is on a row or before a row and that row is deleted, then the cursor is positioned before the row that is immediately after the deleted row. If such a row does not exist, then the cursor is positioned after the last row.

If an error occurs during the execution of an SQL statement that identifies an open cursor, then the effect on the position of that cursor is implementor-defined.

4.3 ERROR TREATMENT

Every executable SQL statement indicates its success or the reason for its failure by a return code. The return code is an integer value in the SMALLINT range.

The return code is communicated via the SQLCODE field of the SQLCA.

There are the following classes of results: "complete success", "success with warning", "no data found" and "error".

The class of result may be determined by checking the contents of the appropriate SQLCA fields or by using a WHENEVER statement (see the section **Whenever Statement**) which also specifies the flow of control.

"Complete success" is indicated by a return code of zero and all warning flags in the SQLCA being set to blank. This means that the statement executed successfully without limitation. The flow of control is unchanged.

"Success with warning" is indicated by a return code of zero and one or more of the warning flags in the SQLCA being set to W, however, see the note below. This means that the statement executed successfully but that the effect was limited by the conditions indicated by the warning flags set. This result class is trapped by a WHENEVER SQLWARNING statement.

"No data found" is indicated by a return code of +100, however, see the note below. This means that the statement executed successfully but that there were no rows that satisfied the statement or there were no more rows to fetch; consequently, no changes were made to the database. This result class is trapped by a WHENEVER NOT FOUND statement.

"Error" is indicated by a negative return code, however, see the note below. This means that the statement did not execute successfully and no changes were made to the database. This result is trapped by a WHENEVER SQLERROR statement. Some errors may cause an open cursor to be closed or an active transaction to be terminated with an implicit statement.

Note: Some implementations currently deviate from the error treatment as described above, see the chapter **Implementation-Specific Issues**.

4.4 SQL DECLARE SECTION

FUNCTION

Definition of host variables used in SQL statements.

SYNTAX

EXEC SQL BEGIN DECLARE SECTION *sql-terminator*
 [*host-variable-definition*]...

EXEC SQL END DECLARE SECTION *sql-terminator*

DESCRIPTION

A host program may contain zero, one or more SQL declare sections.

Every host variable, including indicator variables, referenced by an SQL statement must be declared in exactly one SQL declare section and must be declared textually prior to the referencing SQL statement. (Note that for host variables referenced by a cursor specification, the scope rule applies to the OPEN statements rather than the DECLARE CURSOR statement.)

The header, BEGIN DECLARE SECTION, and the trailer, END DECLARE SECTION, are separately delimited as shown.

Apart from the constraints described below, host variables declared in SQL declare sections are normal host program variables and may be referenced by host program statements without restriction.

Special Rules for COBOL

- The declare section must be placed in the Working-Storage or Linkage Section of the Data Division.
- The declare section must not contain any COPY statements.
- Each declared variable must be an elementary data item with a level number of 01 or 77.
- The following clauses are allowed in the description of a data item:
 PICTURE, SIGN, SYNCHRONIZED, USAGE, VALUE.
- The allowed variants of PICTURE and USAGE are defined below. There are no restrictions with regard to SIGN, SYNCHRONIZED and VALUE.
- The following clauses are not allowed in the description of a data item:
 BLANK, JUSTIFIED, OCCURS, REDEFINES.

Special Rules for C

- The declare section may be placed anywhere C variables may be declared.
- The declare section must not contain any include directives.
- A C variable must not be a pointer variable, except for the case of a C *char* variable, and must not be part of a structure or union.
- The storage classes *auto*, *static* or *extern* may be specified.

4.5 THE SQL COMMUNICATION AREA (SQLCA)

4.5.1 INCLUDE SQLCA FUNCTION

Include the SQLCA declaration in the host program.

SYNTAX

```
EXEC SQL  
    INCLUDE SQLCA  
sql-terminator
```

DESCRIPTION

The SQLCA is the data area in which information is returned about the results of executing an (executable) SQL statement. This data area is included in the host program by using the INCLUDE SQLCA declarative which causes the inclusion of appropriate source statements that define the SQLCA.

Special Rules for COBOL

- The INCLUDE SQLCA declarative must be placed in the Working-Storage or Linkage section of the Data Division.
- The SQLCA is defined by a record description (that is, a non-elementary 01 level data item). The name of the record is SQLCA and all its components have upper case names.
- If the INCLUDE SQLCA declarative is placed in the Linkage Section, the Procedure Division header must contain a USING phrase that includes a parameter named SQLCA.
- For additional placement rules, see the section **Separate Compilation**.

Special Rules for C

- The INCLUDE SQLCA declarative must be placed outside all functions of the C program and must textually precede all SQL statements.
- The SQLCA is defined by a structure. The name of the structure is *sqlca* and all of its components have lower case names.

4.5.2 Contents of the SQLCA

The SQLCA comprises at least the following fields:

SQLCODE

This is a signed four-byte integer field that contains a return code in the SMALLINT range pertaining to the most recently executed SQL statement.

Return code values are defined as follows:

0 (zero)

The statement executed successfully.

+100

The statement executed successfully but there was no data to process. This occurs in the following situations:

FETCH statement

No row to be fetched.

SELECT statement

The result table is empty.

INSERT, *searched DELETE* or *searched UPDATE* statement

The search condition is not satisfied.

Negative values

An error occurred.

SQLERRD

This is an array of six four-byte integers which provide additional information about the execution of the statement. Only the third value is pertinent to this specification.

SQLERRD(3) (or `sqlerrd[2]` for C)

Contains a row count as follows:

After an *INSERT* statement, the number of rows inserted into the table.

After a *searched UPDATE* statement, the number of rows updated in the table.

After a *searched DELETE* statement, the number of rows deleted from the table.

After other statements its contents are implementor-defined.

SQLWARN flags

This is a sequence of at least eight single-character variables that denote warnings. Only the first four are pertinent to this specification.

SQLWARN0

If blank, all other *SQLWARNn* are also blank. If set to *W*, at least one of the other warning characters has been set to *W*.

SQLWARN1

If *W*, at least one character string value was truncated when it was stored into a host variable.

SQLWARN2

If *W*, at least one null value was eliminated from the argument set of a function.

SQLWARN3

If *W*, the number of host variables specified in a *SELECT* or *FETCH* statement is unequal to the number of columns in the table being operated on by the statement.

4.6 DECLARE CURSOR

FUNCTION

Define a cursor.

SYNTAX

EXEC SQL

```
DECLARE cursor-name CURSOR
  FOR query-specification [UNION query-specification]...
  [ ORDER BY sort-specification [, sort-specification]...
  | FOR UPDATE OF column-name [, column-name]... ]
```

sql-terminator

where *sort-specification* is defined as

{ *unsigned-integer* | *column-name* } [ASC | DESC]

DESCRIPTION

A DECLARE CURSOR statement defines and names a cursor for subsequent use within the host program. Cursors are described in the section **Cursors**.

Several cursors may be defined within a host program but they must have distinct names. The DECLARE CURSOR statement defining a cursor must be specified in the source program prior to any SQL statements that reference that cursor.

The derived table defined by the cursor specification is referred to as the result table of the cursor. If UNION is not specified, the result table of the cursor is the table resulting from the *query-specification*. If UNION is specified, the result table of the cursor is the table derived by a merge of the tables resulting from the *query-specifications* followed by an elimination of any duplicate rows. When UNION is specified, the columns of the result table are unharmed.

If UNION is specified, the tables derived from the *query-specifications* must have identical descriptions, except that corresponding column names need not be the same. In this context, identical is defined to mean that corresponding columns match in the following respects:

- i. Each column is a named column. That is, each column is derived from an *expression* consisting of a single *column-name*.
- ii. Both columns either allow or disallow null values.
- iii. The columns have identical named data types (for example, INTEGER). Furthermore, if both columns have data type CHAR, the length must be the same in each case, and if both columns have data type DECIMAL, both the precision and scale must be the same in each case.

The result table of the cursor is updatable if neither ORDER BY nor UNION is specified and the *query-specification* is updatable (see the section **Query Specification**), however, if the cursor is to be referred to by positioned UPDATE or positioned DELETE statements it is also necessary for the FOR UPDATE clause to be specified. The columns identified by the FOR UPDATE clause must be columns of

the table referenced by the (single) FROM clause of the *query-specification* (but need not be columns of the result table of the cursor). The identified columns are the only columns that are allowed to be updated by positioned UPDATE statements. Even though a positioned DELETE statement is not column specific, the execution of such a statement requires a FOR UPDATE clause with at least one valid column name to be specified.

If ORDER BY is not specified, then the ordering of rows of the active set is implementor-defined.

If ORDER BY is specified, the rows of the active set are ordered by the values of the identified columns. If more than one column is identified, the rows are ordered first by the values of the first column, then by the values of the second column, and so on. A column is identified either by its name or by an unsigned integer that represents its ordinal position within the result table. A named column may be identified either by its name or by its ordinal position. An unnamed column must be identified by its ordinal position.

If ASC is specified for a column, that column is ordered in ascending order of its values. If DESC is specified for a column, that column is ordered in descending order of its values. If neither ASC nor DESC is specified, ASC is assumed.

Ordering is determined by the comparison rules defined in the section **Assignments and Comparisons**. The order of the null value relative to non-null values is implementor-defined but must be either greater than or less than all non-null values. If the ordering specification does not encompass all columns of the result table, rows with duplicate values of all identified columns have an undefined order.

No code is generated for a DECLARE CURSOR statement, instead the cursor specification is effectively executed each time the defined cursor is opened, which is also the point at which any referenced host variables are evaluated.

Some implementations currently do generate code for a DECLARE CURSOR statement, see the chapter **Implementation-Specific Issues**.

4.7 WHENEVER STATEMENT

FUNCTION

Specify the action to be taken when an exception condition occurs as a result of executing an SQL statement.

SYNTAX

EXEC SQL

```
WHENEVER {SQLERROR | SQLWARNING | NOT FOUND}  
        {GOTO host-label | CONTINUE}
```

sql-terminator

DESCRIPTION

A WHENEVER statement specifies the action to be taken whenever the result of executing an SQL statement falls into a particular result class. That is to say, if the execution of the SQL statement results in the specified condition, the specified action is performed.

The keywords NOT FOUND, SQLERROR and SQLWARNING correspond to the result classes "no data found", "error" and "success with warning", respectively, as specified in the section **Error Treatment**. Thus, the conditions that may be specified can be summarised as follows:

NOT FOUND

Execution successful but no data found. The SQLCODE value is +100.

SQLERROR

Execution unsuccessful.

SQLWARNING

Execution successful but limited as indicated by the SQLWARN flags.

The actions that may be specified are defined as follows:

CONTINUE

No action is to be taken based on the associated condition. The flow of control remains unchanged.

GOTO *host-label*

Control is to be transferred to *host-label* when the associated condition occurs. For COBOL, *host-label* must be a section-name or unqualified paragraph-name. For C, *host-label* must be a label defined in the same program block or in a block of a higher scope.

Once a WHENEVER statement is declared in the source program, it remains in force until another WHENEVER statement specifying the same condition is declared, and the executable SQL statements within its scope are all those specified between the two occurrences. This means that a WHENEVER statement must textually precede any executable SQL statement it is to affect. If no WHENEVER statement is in force for a particular condition then, by default, the action for that condition is to continue.

No code is generated for a WHENEVER statement at the point at which it is declared, instead appropriate code is effectively generated immediately following those executable SQL statements within its scope.

4.8 SEPARATE COMPILATION

An embedded SQL host program may consist of one or more compilation units written in the same host language.

The scope of a cursor declaration is limited to a single compilation unit. However, the same *cursor-name* must not be used in more than one unit of a host program.

The scope of an exception declarative is limited to a single compilation unit.

Every compilation unit may include zero or more declare sections. A host variable can be used in SQL statements only in that compilation unit in which it is declared.

Every compilation unit must contain an INCLUDE SQLCA declarative. For COBOL, the placement, must be such that all declarations refer to the same common object. For C, the implementor-defined storage class should be such that all declarations refer to the same common object.

The flow of control may freely cross the boundaries of compilation units. Crossing a unit boundary has no effect on transactions or the states of cursors.

Any limits for resource usage (e.g. the limit on the number of open cursors) apply to the host program as a whole, not to the individual compilation units.

Executable SQL Statements

An executable SQL statement is either a data definition statement, a data manipulation statement or a transaction control statement.

5.1 DATA DEFINITION STATEMENTS

A data definition statement modifies the schema contents of the database. A CREATE TABLE statement creates a base table. A CREATE VIEW statement creates a viewed table. A CREATE INDEX statement creates a base table index. A DROP TABLE statement destroys a base table. A DROP VIEW statement destroys a viewed table. A DROP INDEX statement destroys a base table index. An ALTER TABLE statement adds one or more columns to the description of a base table. A GRANT statement grants privileges on base tables and viewed tables. A REVOKE statement revokes privileges on base tables and viewed tables.

For correct processing of the data manipulation statements, it is required that the schema contents of the database referenced by these statements are available at the time of (pre-)compilation. The effects of changing the schema contents after (pre-)compilation are implementor-defined.

Some implementations currently restrict the use of qualification by user-name as defined in the section **Language Structure**. See the chapter **Implementation-Specific Issues**.

5.1.1 ALTER TABLE FUNCTION

Add one or more new columns at the right hand side of an existing base table.

SYNTAX

```
ALTER TABLE base-table-name
  ADD ( column-identifier data-type
      [, column-identifier data-type]... )
```

DESCRIPTION

The ALTER TABLE statement adds one or more new columns to the base table identified by *base-table-name*. The *column-identifiers* specified in the ADD clause must all be unique and must not identify any existing columns in the base table.

The columns are created with data types identified by the corresponding *data-type*. The added columns will allow null values.

After execution of the ALTER TABLE statement every existing row of the base table will have the null value as the value of the added columns.

To add columns to an existing base table, the current user must be its owner.

5.1.2 CREATE INDEX

FUNCTION

Create an index on one or more columns of an existing base table. Ensure uniqueness of the values in the specified columns.

SYNTAX

```
CREATE [UNIQUE] INDEX index-name
    ON base-table-name ( column-identifier [ASC | DESC]
        [, column-identifier [ASC | DESC] ]... )
```

DESCRIPTION

The CREATE INDEX statement creates an index identified by *index-name* on an existing base table identified by *base-table-name*. The *index-name* must not identify an existing index.

The index key is constructed of columns from the specified base table in the given order of significance. When ASC is specified, the order of the referenced column is ascending. When DESC is specified, the order of the referenced column is descending. The default order is ascending.

UNIQUE indicates that at most one row is allowed in the table for each combination of values in the specified columns. For the purpose of this clause two null values are considered equal. The constraint is enforced when rows are inserted or updated and checked during execution of the CREATE INDEX statement.

In order to create an index, the current user must be the owner of the base table and becomes the owner of the index.

5.1.3 CREATE TABLE

FUNCTION

Create a new base table.

SYNTAX

```
CREATE TABLE base-table-name
    ( column-identifier data-type [NOT NULL]
        [, column-identifier data-type [NOT NULL] ]... )
```

DESCRIPTION

The CREATE TABLE statement creates a new base table identified by *base-table-name*. There must not already exist a base table or viewed table identified by *base-table-name*.

The table is owned by the user issuing the CREATE TABLE statement. It is implementor-defined which users may issue the statement.

The created base table has columns identified by the unique *column-identifiers*. The columns are created with data types identified by the corresponding *data-type*.

The NOT NULL clause indicates that null values are not to be allowed in the column. If not specified, the default is to allow null values in the column.

The owner of the table has all privileges on the table. The owner can grant privileges, however, privileges cannot be revoked from the owner.

5.1.4 CREATE VIEW

FUNCTION

Define a viewed table.

SYNTAX

```
CREATE VIEW viewed-table-name
  [( column-identifier [, column-identifier]... ) ]
  AS query-specification
```

DESCRIPTION

The CREATE VIEW statement creates a viewed table identified by *viewed-table-name*. The *viewed-table-name* must not identify an existing viewed table or base table.

The current user must have SELECT privilege on every table in the FROM clause of the *query-specification*. If the SELECT privilege is revoked at any time on any table in the FROM clause of the *query-specification* the viewed table will be dropped automatically.

The viewed table is owned by the user issuing the CREATE VIEW statement.

If *column-identifiers* are specified, the columns of the viewed table will have the specified names. When *column-identifiers* are specified, the number of names specified must be the same as the degree of the derived table defined by the *query-specification* and the same name must not be specified more than once.

If *column-identifiers* are not specified, the columns of the viewed table will have the same names as the corresponding columns of the derived table defined by the *query-specification*. If any two result columns of the *query-specification* have the same name or if any result column is unnamed, *column-identifiers* must be specified for the viewed table.

The data types of the columns of the viewed table are the same as those of the corresponding result columns of the *query-specification*.

The viewed table is updatable if the derived table defined by the *query-specification* is updatable. (See the section **Query-specifications**.)

The owner will have INSERT, DELETE or UPDATE privileges on the viewed table if the viewed table is updatable and if the owner has that privilege on the table from which the viewed table is derived. Otherwise the owner will only have SELECT privilege on the viewed table.

If the *query-specification* contains a GROUP BY clause, *viewed-table-name* identifies a grouped view. A grouped view is a set of groups derived during the evaluation of the GROUP BY clause. A group is a multi-set of rows in which all values of the grouping columns are equal.

The *query-specification* defining a viewed table must not reference any host variables.

The owner of the viewed table must also be the owner of the base table to grant any privileges on the viewed table.

5.1.5 DROP INDEX

FUNCTION

Destroy an index.

SYNTAX

DROP INDEX *index-name*

DESCRIPTION

The index identified by *index-name* is removed from the database.

The current user must be the owner of the specified index to issue the DROP INDEX statement.

5.1.6 DROP TABLE

FUNCTION

Destroy a table.

SYNTAX

DROP TABLE *base-table-name*

DESCRIPTION

The table identified by *base-table-name* is removed from the database.

The current user must be the owner of the base table to issue the DROP TABLE statement.

When a base table is dropped, all indexes and viewed tables defined on that base table are dropped. All privileges granted on it are revoked.

5.1.7 DROP VIEW

FUNCTION

Destroy a view.

SYNTAX

DROP VIEW *viewed-table-name*

DESCRIPTION

The viewed table identified by *viewed-table-name* is removed from the database.

The current user must be the owner of the viewed table to issue the DROP VIEW statement.

All viewed tables based on a dropped viewed table are dropped as well. All privileges granted on it are revoked.

5.1.8 GRANT

FUNCTION

Grant privileges to other users.

SYNTAX

GRANT {ALL | *privilege* [, *privilege*]... } ON *table-name*
TO {PUBLIC | {*user-name* [, *user-name*]... } }

where *privilege* is one of the following:

SELECT
DELETE
INSERT
UPDATE [(*column-identifier* [, *column-identifier*]...)]

DESCRIPTION

The GRANT statement grants the specified *privileges* on the table identified by *table-name* to some other users.

The current user must be the owner of the table identified by *table-name* to issue the GRANT statement.

SELECT specifies the right to retrieve values. DELETE specifies the right to delete rows. INSERT specifies the right to insert rows. UPDATE specifies the right to update the columns identified by *column-identifiers*. If no columns are specified, UPDATE specifies the right to update all columns.

The SELECT *privilege* can be granted for base tables and for viewed tables. The INSERT, DELETE or UPDATE *privilege* can only be granted for base tables and updatable viewed tables.

ALL specifies every applicable *privilege* described above with no restriction on UPDATE.

For additional restrictions with respect to viewed tables, see the description of the CREATE VIEW statement.

The option PUBLIC identifies all users, whereas specific users are identified by providing *user-names*.

Each *privilege* that is specified explicitly (or implicitly using ALL) must not already have been granted to any *user-name* (nor to PUBLIC if that option is specified). The same *privilege* or the same *user-name* must not be specified more than once.

5.1.9 REVOKE FUNCTION

Revoke privileges previously granted to other users.

SYNTAX

```
REVOKE {ALL | privilege [, privilege]... } ON table-name  
FROM {PUBLIC | {user-name [, user-name]... } }
```

where *privilege* is one of the following:

```
SELECT  
DELETE  
INSERT  
UPDATE
```

DESCRIPTION

The REVOKE statement revokes the specified *privileges* on the table identified by *table-name* from some other users.

The current user must be the owner of the table identified by *table-name* to issue the REVOKE statement.

SELECT specifies the right to retrieve values. DELETE specifies the right to delete rows. INSERT specifies the right to insert rows. UPDATE specifies the right to update columns.

If ALL is not specified, the current user must have granted all the specified *privileges* on the identified table to all identified users. If ALL is specified, every *privilege* granted by the current user on the identified table, and there must be at least one such *privilege*, is revoked from all the identified users. The same *privilege* or the same *user-name* must not be specified more than once.

The option PUBLIC identifies all users, whereas specific users are identified by providing *user-names*. The PUBLIC option can only be used to revoke those *privileges* previously granted by a GRANT statement with the PUBLIC option.

The REVOKE statement will cause the drop of a viewed table if it revokes a SELECT *privilege* needed for the creation of that viewed table.

User note:

In contrast to the GRANT statement, the UPDATE *privilege* is not column specific.

To revoke UPDATE *privilege* for less than all columns of a table, first revoke the UPDATE *privilege* on all columns and then GRANT again the UPDATE *privilege* on those columns for which the *privilege* should be kept.

5.2 DATA MANIPULATION STATEMENTS

A data manipulation statement operates on the data contents of the database or controls the state of a cursor. An OPEN statement opens a cursor. A CLOSE statement closes a cursor. A SELECT or FETCH statement fetches values from a table. An INSERT statement inserts rows into a table. An UPDATE statement updates the values in rows of a table. A DELETE statement deletes rows of a table.

The reader is reminded that the term table refers to viewed tables as well as base tables.

Cursors are described in the section **Cursors**.

The assignment and comparison rules are described in the section **Assignments and Comparisons**.

5.2.1 CLOSE FUNCTION

Close a cursor.

SYNTAX

CLOSE *cursor-name*

DESCRIPTION

The cursor identified by *cursor-name* is placed in the closed state, and the active set associated with the cursor is no longer accessible. The cursor must be in the open state.

5.2.2 DELETE positioned FUNCTION

Delete the current row of an active set.

SYNTAX

DELETE FROM *table-name*
WHERE CURRENT OF *cursor-name*

DESCRIPTION

The row from which the current row of the active set of the cursor identified by *cursor-name* is derived, is deleted. If the table identified by *table-name* is a viewed table, the row that is deleted is the corresponding row of the base table from which it is derived.

The current user must have DELETE privilege on *table-name* to issue the positioned DELETE statement.

The cursor specification for *cursor-name* must include the FOR UPDATE clause and the result table of the cursor must be updatable.

The table identified by *table-name* must be the (single) table referenced by the FROM clause of the *query-specification* that defines the result table of the cursor.

The cursor must be positioned on a row in its active set.

5.2.3 DELETE searched

FUNCTION

Delete rows of a table.

SYNTAX

DELETE

FROM *table-name*

[WHERE *search-condition*]

DESCRIPTION

Zero or more rows are deleted from the table identified by *table-name*. If the identified table is a viewed table, the rows that are deleted are the corresponding rows of the base table from which it is derived.

The current user must have DELETE privilege on *table-name* to issue the searched DELETE statement.

The table identified by *table-name* must be updatable and not referenced by a FROM clause of any *sub-query* contained in the *search-condition*.

The scope of the *table-name* is the entire searched DELETE statement.

If *search-condition* is not specified, then all rows of the table identified by *table-name* are deleted.

If *search-condition* is specified, then it is applied to each row of the table identified by *table-name*, and all rows for which the result of the *search-condition* is true are deleted. If no row satisfies the *search-condition*, SQLCODE is set to indicate that there were no rows to delete.

Each *sub-query* in the *search-condition* is effectively executed for each row of the table identified by *table-name*, and the results used in the evaluation of the *search-condition* for that row.

If any executed *sub-query* contains a reference to a column of the table identified by *table-name* (that is, a correlated reference), the reference is to the value of that column in the given row.

The number of rows deleted by the execution of the statement is indicated by the value of SQLERRD(3).

5.2.4 FETCH

FUNCTION

Advance the cursor to the next row of the active set and assign the column values to host variables.

SYNTAX

```
FETCH cursor-name  
    INTO host-variable-reference [, host-variable-reference]...
```

DESCRIPTION

The cursor identified by *cursor-name* is advanced to the next row of its active set and the column values are assigned to host variables.

The identified cursor must be in the open state.

The number of *host-variable-references* must be equal to the degree of the result table of the cursor, otherwise SQLWARN0 and SQLWARN3 are set to W.

If the active set is empty, or the cursor is positioned on or after the last row, SQLCODE is set to indicate that there was no row to retrieve, and no values are assigned to the host variables identified by the *host-variable-references*.

If an error occurs during assignment to any of the identified host variables the contents of all identified host variables are implementor-defined.

5.2.5 INSERT

FUNCTION

Insert rows into a table.

SYNTAX

```
INSERT INTO table-name [(column-identifier [, column-identifier]... )]  
    { query-specification  
    | VALUES (insert-value [, insert-value]... ) }
```

where *insert-value* is defined as:

```
host-variable-reference | literal | NULL
```

DESCRIPTION

Zero or more rows are inserted into the table identified by *table-name*. If the identified table is a viewed table, the rows are inserted into the base table from which it is derived.

The current user must have INSERT privilege on *table-name* to issue the INSERT statement.

The table identified by *table-name* must be updatable and not referenced by a FROM clause of the *query-specification* or of any *sub-query* contained in the *query-specification*.

Each *column-identifier* must identify a column of the table identified by *table-name*, and the same column must not be identified more than once. Omission of *column-identifier* is an implicit specification of all columns of the table identified by *table-name*, in the ascending sequence of their ordinal position within that table.

In the VALUES form the number of *insert-values* must be equal to the number of *column-identifiers* and one row consisting of these values is inserted.

In the *query-specification* form, the degree of the derived table must be equal to the number of *column-identifiers* and the cardinality of the derived table is the number of rows inserted. If the derived table is empty, SQLCODE is set to indicate that there were no rows to insert.

The table identified by *table-name* can be a base table or a viewed table derived from a base table.

The null value is the *insert-value* of any column of the table not identified in the column list. Furthermore, if a viewed table is specified, the null value is the *insert-value* of any column of its base table not included in the viewed table. If the *insert-value* of any column is null, the column must be defined to allow null values.

The value of the *i*th column of a row in the derived table, or the *i*th *insert-value* in the VALUES form, is assigned to the base table column corresponding to the *i*th column of the table identified by *table-name* or to the column identified by the *i*th *column-identifier*.

The number of rows inserted by the execution of the statement is indicated by the value of SQLERRD(3).

5.2.6 OPEN

FUNCTION

Open a cursor.

SYNTAX

OPEN *cursor-name*

DESCRIPTION

The cursor specification for *cursor-name* is effectively executed using the current values of any referenced host variables, see the section **Declare Cursor**. In this way a multi-set of records is identified, and becomes the active set for this cursor.

The current user must have SELECT privilege on every table referenced by the cursor specification for *cursor-name*.

The cursor must be in the closed state.

The cursor is placed in the open state, and its position is before the first row in the active set. While the cursor is in the open state, subsequent changes to the values of any referenced host variables will not affect the active set.

5.2.7 SELECT

FUNCTION

Assign the values from the specified table to host variables.

SYNTAX

```
SELECT [ ALL | DISTINCT ] { * | expression [, expression]... }  
      INTO host-variable-reference [, host-variable-reference]...  
      FROM table-reference [, table-reference]...  
      [ WHERE search-condition ]
```

DESCRIPTION

The SELECT statement specifies a one-row table and assigns its values to host variables.

The current user must have SELECT privilege on the table(s) referenced by the FROM clause to issue the SELECT statement.

If the SELECT clause contains the * character, the result columns are all the columns of the tables referenced by the FROM clause; if DISTINCT is specified, any redundant duplicate rows are eliminated from the result table.

The number of result columns (*expressions*) specified in the SELECT clause must be equal to the number of *host-variable-references* in the INTO clause, otherwise SQLWARN0 and SQLWARN3 are set to W.

The cardinality of the table specified by the SELECT statement must not be greater than one; if the cardinality is zero (empty table), SQLCODE is set to indicate that there was no row to retrieve.

If the resulting table is not empty, values in the row of this table are assigned to their corresponding *host-variable-references*: the *i*th *host-variable-reference* is assigned the value of the *i*th column of the result table.

If an error occurs during assignment to any of the identified host variables the contents of all identified host variables are implementor-defined.

5.2.8 UPDATE positioned

FUNCTION

Update the current row of an active set

SYNTAX

```
UPDATE table-name  
      SET column-identifier = { expression | NULL }  
      [, column-identifier = { expression | NULL } ]...  
      WHERE CURRENT OF cursor-name
```

DESCRIPTION

The row from which the current row of the active set of the cursor identified by *cursor-name* is derived, is updated. If the table identified by *table-name* is a viewed table, the updates are applied to the corresponding row of the base table from which it is derived.

Column-identifier identifies a column to be updated (an update column) and must identify a column of the table identified by *table-name* (the update table). The same column must not be identified more than once.

The current user must have UPDATE privilege on each update column to issue the positioned UPDATE statement.

The cursor specification for *cursor-name* must include the FOR UPDATE clause and the result table of the cursor must be updatable. The columns identified by the FOR UPDATE clause must include the update columns.

The update table must be the (single) table referenced by the FROM clause of the *query-specification* that defines the result table of the cursor. The scope of the *table-name* is the entire positioned UPDATE statement.

Expression or NULL represents the update value of the corresponding update column. That is, for each update column, the value of that column in the row to be updated is replaced by the null value or the value of the corresponding *expression*.

If *expression* is specified, it must not include a *set-function-reference* and any column it references must be a column of the update table. The value specified by such a referenced column is the value of the identified column in the row to be updated before any values in that row are updated.

If the update value of a column is the null value, the corresponding update column must be defined to allow null values.

The cursor must be positioned on a row in its active set.

5.2.9 UPDATE searched

FUNCTION

Update values of columns in a table.

SYNTAX

UPDATE *table-name*

```
SET column-identifier = { expression | NULL }  
[, column-identifier = { expression | NULL } ]...  
[WHERE search-condition]
```

DESCRIPTION

Zero or more rows of the table identified by *table-name* are updated. If the identified table is a viewed table, the updates are applied to the corresponding rows of the base table from which it is derived.

Column-identifier identifies a column to be updated (an update column) and must identify a column of the table identified by *table-name* (the update table). The same

column must not be identified more than once.

The current user must have UPDATE privilege on each update column to issue the searched UPDATE statement.

The update table must be updatable and must not be referenced by the FROM clause of any *sub-query* contained in the *search-condition*. The scope of the *table-name* is the entire searched UPDATE statement.

Expression or NULL represents the update value of the corresponding update column. That is, for each update column, the value of that column in the row to be updated is replaced by the null value or the value of the corresponding *expression*.

If *expression* is specified, it must not include a *set-function-reference* and any column it references must be a column of the update table. The value specified by such a referenced column is the value of the identified column in the row to be updated before any values in that row are updated.

If the update value of a column is the null value, the corresponding update column must be defined to allow null values.

If *search-condition* is not specified, all rows of the update table are updated.

If *search-condition* is specified, it is applied to each row of the update table and all rows for which the result of the *search-condition* is true are updated. If no row satisfies the *search-condition*, SQLCODE is set to indicate that there were no rows to update.

Each *sub-query* in the *search-condition* is effectively executed for each row of the update table and the results used in the evaluation of the *search-condition* for that row.

If any executed *sub-query* contains a reference to a column of the update table (that is, a correlated reference), the reference is to the value of that column in the given row.

The number of rows updated by the execution of the statement is indicated by the value of SQLERRD(3).

5.3 TRANSACTION CONTROL STATEMENTS

A transaction control statement terminates an active transaction. A COMMIT statement commits the database changes made by the terminated transaction. A ROLLBACK statement backs out the database changes made by the terminated transaction.

5.3.1 COMMIT

FUNCTION

Successfully terminate the current transaction.

SYNTAX

COMMIT WORK

DESCRIPTION

The current transaction is terminated. Any cursors that were opened inside that terminated transaction are closed. Any changes to the database that were made by that terminated transaction are committed.

It is implementor-defined whether this statement will initiate another transaction.

5.3.2 ROLLBACK

FUNCTION

The current transaction is cancelled.

SYNTAX

ROLLBACK WORK

DESCRIPTION

The current transaction is terminated. Any cursors that were opened inside that terminated transaction are closed. Any changes to the database that were made by that terminated transaction are cancelled.

It is implementor-defined whether this statement will initiate another transaction.

Implementation-Specific Issues

6.1 LIMITS

Limits vary between implementations. This section defines the maximum values which application developers may safely assume will be supported on all X/OPEN systems. For example, the maximum length of a character string varies between 240 and 4096. For total portability, developers should assume that the maximum length is 240 characters.

- a) There is a limit on the length of a character string. This limit is not less than two hundred and forty (240).
- b) There is a limit on the precision of any DECIMAL numeric type. This limit is not less than fifteen (15).
- c) The precision of an INTEGER number is ten (10). The precision of a SMALLINT number is five (5). The precision of a FLOAT number is fifteen (15).
- d) There is a limit on the number of columns in a table. This limit is not less than one hundred (100).
- e) There is a limit on the total length of a row. One may safely assume that if, for a table, the sum of
 - (1) twice the number of columns in the table,
 - (2) the sum of the lengths of all character fields in a row,
 - (3) the sum of the precisions of all numeric fields in a row,does not exceed two thousand (2000), the limit on the length of a row is not exceeded.
- f) There are limits on the columns constituting an index. The limit on the number of columns is not less than six (6). The limit on the total length of an index key is not less than one hundred and twenty (120) using the same algorithm as defined in e) above.
- g) There are limits on the columns specified in a GROUP BY clause. These limits are the same as those for an index, see f) above.
- h) There are limits on the columns specified in an ORDER BY clause. These limits are the same as those for an index, see f) above.
- i) There is a limit on the number of tables directly or indirectly referenced in a statement. This limit is not less than ten (10).
- j) There is a limit on the number of cursors simultaneously open. This limit is not less than ten (10).

6.2 RESTRICTIONS ON NAMES

- a) Most implementations have reserved words in addition to those specified in the section **Keywords**. To avoid any problems when moving onto a different implementation, all programs should avoid using identifiers which might be such keywords. This can be done, for example, by including underscore characters in the identifiers.

The following list identifies known additional keywords defined by a number of products widely available when Issue 2 of the Portability Guide was published.

ABORT	CHR2FLO	DBYTE
ABS	CHR2FLOA	DEFAULT
ACTIVATE	CHR2FLOAT	DEFER
ADDFORM	CHR2INT	DEFINE
AFTER	CLEAR	DEFINITION
APPEND	CLEARROW	DELETEROW
ARCHIVE	CLUSTER	DESCRIBE
ASCII	CLUSTERING	DESCRIPTOR
ASSERT	COLD	DESTPOS
ASSIGN	COLUMN	DEVICE
AT	COMMAND	DEVSPACE
ATTRIBUTES	COMMENT	DIRECT
AUDIT	COMPRESS	DISCONNECT
AUTONEXT	CONCAT	DISPLACE
AVGU	COND	DISPLAY
BACKOUT	CONFIG	DISTRIBUTION
BEFORE	CONNECT	DIV
BEGINLOAD	CONTAIN	DOES
BEGINMODIFY	CONTAINS	DOMAIN
BEGINNING	COPY	DOWN
BEGWORK	COUNTU	DUAL
BREAK	CRASH	DUPLICATES
BREAKDISPLAY	DATABASE	EACH
BUFFER	DATAPAGES	EBCDIC
BUFFERED	DATE	EDITADD
BULK	DAY	EDITUPDATE
BYTE	DAYNUM	ELSE
CALL	DBA	ENDDATA
CANCEL	DBE	ENDDISPLAY
CASCADE	DBEFILE	ENDFORMS
CHANGE	DBEFILE0	ENDIF
CHECKPOINT	DBEFILESET	ENDING
CHR2FL	DBSPACE	ENDLOAD

ENDLOOP	INIT	MONEY
ENDMODIFY	INITIAL	MONITOR
ENDPOS	INITIALIZE	MONTH
ERASE	INITTABLE	MOVE
EVALUATE	INPUT	MULTI
EVERY	INQUIRE_FRS	NAME
EXCLUSIVE	INSERTROW	NEW
EXECUTE	INSTRUCTIONS	NEWLOG
EXPLICIT	INT2CHR	NEXT
FIELD	INTEGRITY	NO
FILE	INTERSECT	NOCOMPRESS
FINALIZE	ISAM	NOJOURNALING
FINDSTR	JOURNALING	NOLOG
FIRST	KEY	NORMAL
FIRSTPOS	LAST	NOSYSSORT
FIXED	LASTPOS	NOTRANS
FL	LENGTH	NOWAIT
FORMAT	LENSTR	NULLIFY
FORMDATA	LEVEL	NULLVAL
FORMINIT	LINK	NUMBER
FORMS	LIST	NXFIELD
FRS	LOAD	OFF
GET	LOADTABLE	OLD
GETFORM	LOCAL	ONTO
GETOPER	LOCATION	OPTIMIZE
GETROW	LOCK	OPTIONS
GLOBAL	LOCKING	OUT
GRAPHIC	LOG	OUTER
HEADER	LONG	OUTPUT
HELPPFILE	LOWER	OWNER
HELP_FRS	LPAD	OWNERSHIP
HOLD	MATCHES	PAGE
IDENTIFIED	MAXEXTENTS	PAGES
IF	MAXRECLEN	PARAM
IFDEF	MDY	PARTITION
IGNORE	MENUITEM	PASSWORD
IMAGE	MESSAGE	PATTERN
IMMEDIATE	MFETCH	PCTFREE
IMPLICIT	MINRECLEN	PERMIT
INCREMENT	MINUS	PLACE
INDEXED	MIXED	POS
INDEXNAME	MOD	POWER
INDEXPAGES	MODE	PREPARE
INFO	MODIFY	PRESERVE

PREV	ROWS	SUPERDBA
PRINT	RPAD	SYNONYM
PRINTSCREEN	RUN	SYSDATE
PRIOR	SAMPLSTDEV	SYSSORT
PRIV	SAVE	TABLEDATA
PRIVATE	SAVEPOINT	TEMP
PROGRAM	SCREEN	TERMINATE
PROGUSAGE	SCROLL	THEN
PROMPT	SCROLLEDOWN	TID
PUBLICREAD	SCROLLUP	TIME
PUTFORM	SEARCH	TODAY
PUTOPER	SEGMENT	TOLOWER
PUTROW	SEL	TOUPPER
QUERY	SELE	TRAILER
QUICK	SELEC	TRANS
RANGE	SELUPD	TRANSACTION
READ	SERIAL	TRANSFER
READPASS	SET _FRS	TRIGGER
READWRITE	SHARE	TRUNC
RECONNECT	SHOW	TYPE
RECOVER	SHUTDOWN	UID
REDISPLAY	SIZE	UNBUFFERED
REJECT	SLEEP	UNLOAD
RELEASE	SMALLFLOAT	UNLOADTABLE
RELOAD	SORT	UNLOCK
RELOCATE	SOUNDS	UNTIL
REMOVE	SOURCEPOS	UP
RENAME	SPACE	UPPER
REPLACE	SQLDA	USAGE
REPLSTR	SQLEXCEPTION	USING
RESET	SQLEXPLAIN	VALIDATE
RESOURCE	SQLNOTFOUND	VALIDROW
REST	SQRT	VARC
RESTART	START	VARCH
RESTORE	STARTPOS	VARCHA
RESTRICT	STATE	VARCHAR
RESUME	STATISTICS	VARGRAPHIC
RETRIEVE	STDEV	VERIFY
RIGHT	STOP	VERSION
ROLLFORWARD	STORE	WAIT
ROUND	STRING	WEEKDAY
ROW	SUBMENU	WRITE
ROWID	SUBSTR	WRITEPASS
ROWNUM	SUMU	YEAR

- b) To avoid name conflicts between host programs and the specific run-time routines of an implementation, it is recommended that programs do not use procedure, function or variable names starting with the letters *sql* or *SQL*.
- c) Some implementations have a single namespace per user-name and some have a single namespace per database. To ensure portability of names the following (additional) rules should be adhered to:
 - i. Index-identifiers should be distinct from the name of any base table or viewed table owned by the creator of the index.
 - ii. Base-table-identifiers, viewed-table-identifiers and index-identifiers should each be unique within a database.
- d) To avoid name conflicts between system-defined objects and user-defined objects, it is recommended that user-defined names do not start with the letters *SYS*.
- e) Some implementations have keyword oriented parsers. To ensure portability, the names of embedded host variables should be distinct from SQL keywords (including those specified in a) above).
- f) Some implementations currently do not allow qualification by user-name within table-names and index-names in CREATE statements. In these statements user-name must be the name of the current user, so it is recommended that programs do not use this optional qualification.

6.3 DATA DEFINITION

Several products currently do not implement the semantics of transactions with respect to modifications of the schema contents. Some of them implement data definition statements as separate transactions which are immediately committed. Others also commit data manipulation statements previously executed by the current transaction when a data definition statement is executed. Of course, some products have data definition statements fully controlled by transaction control statements.

If a host program is to be truly portable, it must produce the same effect on any implementation of this definition. To achieve this behaviour, it must currently be programmed following these restrictions:

- a) A transaction should not contain both data manipulation statements and data definition statements.
- b) Every data definition statement should immediately be followed by a COMMIT statement.
- c) A ROLLBACK statement should not be assumed to undo the effects of any data definition statement.
- d) If the program is assumed to be restartable after a crash, it must analyze its previous progress and must be prepared to clean up any intermediate state of data definition operations.

In effect this means that there may currently be two classes of transactions:

- a) Data manipulation transactions which follow the defined rules for transactions and may either be committed or aborted.
- b) Data definition transactions which consist of a single statement only followed by a COMMIT statement.

6.4 ASSIGNMENTS

Some implementations currently do not support the semantics of character string assignments as defined in the section **Assignments and Comparisons** but also set an indicator variable to the original length of the character string if it has been assigned without truncation. It is recommended that programs use the setting of SQLWARN1 to test for truncation.

6.5 ERROR TREATMENT

Some implementations currently do not support the SQLCODE value +100 for the "no data found" condition, but use an implementor-defined positive value instead. However, this result class will always be trapped by a WHENEVER NOT FOUND statement.

Implementations also currently differ in their handling of positive return codes other than +100; some treat them as errors and trap them via SQLERROR and some treat them as warnings and trap them via SQLWARNING. It is therefore recommended that programs use the WHENEVER statement to determine the result class of an exception condition rather than test SQLCODE explicitly.

6.6 DECLARE CURSOR

Some implementations currently generate code for a DECLARE CURSOR statement so it must be logically positioned before any statement that references the defined cursor.

Syntax Summary

This summary uses the notation defined in the section **Embedding SQL Constructs**.

A.1 COMMON ELEMENTS

```
approximate-numeric-literal ::= mantissaEexponent

    mantissa ::= exact-numeric-literal
    exponent ::= [+|-]unsigned-integer

approximate-numeric-type ::= FLOAT

base-table-identifier ::= user-defined-name

base-table-name ::= [user-name.]base-table-identifier

between-predicate ::=
    expression [NOT] BETWEEN expression AND expression

character ::=
    any character in the implementor's character set except
    the newline indication

character-string-literal ::= '{character}...'

character-string-type ::=
    CHAR(length)

    length ::= unsigned-integer

column-identifier ::= user-defined-name

column-name ::=
    [{table-name | correlation-name}.]column-identifier

comparison-operator ::= < | > | <= | >= | = | <>

comparison-predicate ::=
    expression comparison-operator
    { expression | ( sub-query ) }

correlation-name ::= user-defined-name

cursor-name ::= user-defined-name

data-type ::=
    character-string-type
    | exact-numeric-type
    | approximate-numeric-type
```


digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

exact-numeric-literal ::=
 [+|-]{unsigned-integer[.unsigned-integer]
 | unsigned-integer.
 | .unsigned-integer}

exact-numeric-type ::=
 DECIMAL(precision,scale)
 | INTEGER
 | SMALLINT

precision ::= unsigned-integer

scale ::= unsigned-integer

exists-predicate ::= EXISTS (sub-query)

expression ::= term | expression {+|-} term

term ::= factor | term {*/|} factor

factor ::= [+|-]primary

primary ::= column-name

 | host-variable-reference

 | literal

 | set-function-reference

 | USER

 | (expression)

host-variable-reference ::=
 embedded-variable-name[indicator-variable]

embedded-variable-name ::= :host-identifier

indicator-variable ::= :host-identifier

index-identifier ::= user-defined-name

index-name ::= [user-name.]index-identifier

in-predicate ::=

 expression [NOT] IN { (value {, value}...)
 | (sub-query) }

value ::= host-variable-reference | literal | USER

keyword ::=

 see the section Language Structure

letter ::= lower-case-letter | upper-case-letter

like-predicate ::= column-name [NOT] LIKE pattern-value

pattern-value ::=
 character-string-literal
 | host-variable-reference
 | USER

literal ::= character-string-literal | numeric-literal

lower-case-letter ::=
 a | b | c | d | e | f | g | h | i | j | k | l | m
 | n | o | p | q | r | s | t | u | v | w | x | y | z

null-predicate ::= column-name IS [NOT] NULL

numeric-literal ::=
 exact-numeric-literal
 | approximate-numeric-literal

predicate ::=
 between-predicate
 | comparison-predicate
 | exists-predicate
 | in-predicate
 | like-predicate
 | null-predicate
 | quantified-predicate

quantified-predicate ::=
 expression comparison-operator
 {ALL | ANY} (sub-query)

query-specification ::=
 SELECT [ALL | DISTINCT]
 {* | expression [, expression]... }
 FROM table-reference [, table-reference]...
 [WHERE search-condition]
 [GROUP BY column-name [, column-name]...]
 [HAVING search-condition]

search-condition ::=
 boolean-term [OR search-condition]

 boolean-term ::= boolean-factor [AND boolean-term]
 boolean-factor ::= [NOT] boolean-primary
 boolean-primary ::= predicate | (search-condition)

separator ::=

implementor-defined end-of-line indicator
| blank character

set-function-reference ::=

COUNT(*) | distinct-function | all-function

distinct-function ::= {AVG | COUNT | MAX | MIN | SUM}
(DISTINCT column-name)

all-function ::= {AVG | MAX | MIN | SUM}
(expression)

sub-query ::= SELECT [ALL | DISTINCT] { * | expression }
FROM table-reference [, table-reference]...
[WHERE search-condition]
[GROUP BY column-name [, column-name]...]
[HAVING search-condition]

table-identifier ::= user-defined-name

table-name ::= [user-name.]table-identifier

table-reference ::= table-name [correlation-name]

token ::= delimiter-token | non-delimiter-token

delimiter-token ::=

character-string-literal

| , | (|) | < | > | . | :

| = | * | + | - | / | <= | >= | <=

non-delimiter-token ::=

keyword

| numeric-literal

| user-defined-name

| host-identifier

unsigned-integer ::= {digit}...

upper-case-letter ::=

A | B | C | D | E | F | G | H | I | J | K | L | M
N | O | P | Q | R | S | T | U | V | W | X | Y | Z

user-defined-name ::=

letter[digit | letter | _]...

user-name ::= user-defined-name

viewed-table-identifier ::= user-defined-name

viewed-table-name ::= [user-name.]viewed-table-identifier

A.2 EMBEDDED ASPECTS

```
declare-cursor ::=
    DECLARE cursor-name CURSOR FOR
        query-specification [UNION query-specification]...
        [FOR UPDATE OF column-name [, column-name]...
        | order-by-clause]

order-by-clause ::=
    ORDER BY sort-specification [, sort-specification]...
sort-specification ::=
    {unsigned-integer | column-name} [ASC | DESC]

embedded-exception-declaration ::=
    WHENEVER { SQLERROR
        | SQLWARNING
        | NOT FOUND}
    {CONTINUE | GOTO host-label}

embedded-sql-construct ::=
    embedded-sql-declarative | embedded-sql-statement

embedded-sql-declarative ::=
    embedded-sql-declare-section | include-sqlca

embedded-sql-declare-section ::=
    sql-prefix BEGIN DECLARE SECTION sql-terminator
        [host-variable-definition]...
    sql-prefix END DECLARE SECTION sql-terminator

embedded-sql-statement ::=
    sql-prefix
    { declare-cursor
    | embedded-exception-declaration
    | executable-sql-statement}
    sql-terminator
```

```
executable-sql-statement ::=
    alter-table-statement
    | close-statement
    | commit-statement
    | create-index-statement
    | create-table-statement
    | create-view-statement
    | delete-statement-positioned
    | delete-statement-searched
    | drop-index-statement
    | drop-table-statement
    | drop-view-statement
    | fetch-statement
    | grant-statement
    | insert-statement
    | open-statement
    | revoke-statement
    | rollback-statement
    | select-statement
    | update-statement-positioned
    | update-statement-searched

include-sqlca ::= sql-prefix INCLUDE SQLCA sql-terminator

sql-prefix ::= EXEC SQL

sql-terminator ::= END-EXEC for COBOL
                  ; for C
```


A.3 SQL STATEMENTS

```
alter-table-statement ::=
    ALTER TABLE base-table-name
    ADD ( column-identifier data-type
        [, column-identifier data-type]... )

close-statement ::= CLOSE cursor-name

commit-statement ::= COMMIT WORK

create-index-statement ::=
    CREATE [UNIQUE] INDEX index-name
    ON base-table-name
    ( column-identifier [ASC | DESC]
    [, column-identifier [ASC | DESC] ]... )

create-table-statement ::=
    CREATE TABLE base-table-name
    ( column-identifier data-type [NOT NULL]
    [, column-identifier data-type [NOT NULL]]... )

create-view statement ::=
    CREATE VIEW viewed-table-name
    [( column-identifier [, column-identifier]... )]
    AS query-specification

delete-statement-positioned ::= DELETE FROM table-name
                                WHERE CURRENT OF cursor-name

delete-statement-searched ::= DELETE FROM table-name
                                [WHERE search-condition]

drop-index-statement ::= DROP INDEX index-name

drop-table-statement ::= DROP TABLE base-table-name

drop-view-statement ::= DROP VIEW viewed-table-name

fetch-statement ::=
    FETCH cursor-name INTO host-variable-reference
    [, host-variable-reference]...
```

```

grant-statement ::=
    GRANT {ALL | privilege [, privilege]... }
    ON table-name
    TO {PUBLIC | user-name [, user-name]... }

privilege ::=
    DELETE
    | INSERT
    | SELECT
    | UPDATE [( column-identifier
              [, column-identifier]... )]

insert-statement ::=
    INSERT INTO table-name [( column-identifier
                             [, column-identifier]... )]
    { query-specification
    | VALUES ( insert-value [, insert-value]... )}

insert-value ::=
    host-variable-reference
    | literal
    | NULL
    | USER

open-statement ::= OPEN cursor-name

revoke-statement ::=
    REVOKE {ALL | privilege [, privilege]... }
    ON table-name
    FROM {PUBLIC | user-name [, user-name]... }

privilege ::=
    DELETE
    | INSERT
    | SELECT
    | UPDATE

rollback-statement ::= ROLLBACK WORK

select-statement ::=
    SELECT [ALL | DISTINCT]
           { * | expression [, expression]... }
    INTO host-variable-reference
           [, host-variable-reference]...
    FROM table-reference [, table-reference]...
    [WHERE search-condition]
    
```

update-statement-positioned ::=

```
UPDATE table-name
SET column-identifier = {expression | NULL}
[, column-identifier = {expression | NULL}]...
WHERE CURRENT OF cursor-name
```

update-statement-searched ::=

```
UPDATE table-name
SET column-identifier = {expression | NULL }
[, column-identifier = {expression | NULL}]...
[WHERE search-condition]
```


ANS X3.135 Database Language (SQL)

The X/OPEN SQL definition is based upon American National Standard Database Language SQL "X3.135 - 1986"; however, because of X/OPEN's need to tailor its definition to meet a spectrum of existing UNIX implementations and its desire to extend functionality to reflect common usage, there are some differences between the two definitions. This appendix addresses those differences.

B.1 FACILITY LEVEL

X/OPEN includes all the "X3.135-1986" level 1 facilities except where indicated below under **Discrepancies**.

X/OPEN includes the following "X3.135-1986" level 2 facilities:

- Transactions atomic with respect to recovery
- Identifiers containing up to 18 characters
- Table-name qualification by user-name
- The keyword USER
- Indicator variables
- Outer references
- Keyword ALL allowed in query-specifications and sub-queries.
- DISTINCT with AVG, MAX, MIN and SUM
- <> in comparisons
- NOT LIKE
- EXISTS predicate
- Equality of null grouping column values
- The updatable query-specification definition
- NOT NULL for a column
- Specifying +100 rather than an implementor-defined positive value for the SQLCODE value corresponding to the "not found" condition
- Statements atomic with respect to database changes
- Cursor sort order using unsigned integers
- Keyword ASC allowed in defining cursor sort order
- UNION in cursor declaration

- Multi-row INSERT using a query-specification
- UPDATE CURRENT and DELETE CURRENT

X/OPEN excludes the following "X3.135-1986" level 2 facilities:

- Keyword ALL allowed in set functions
- Escape characters in the LIKE predicate
- Explicit schema and associated authorisation-identifier (however, see **Extensions**).
- Unique constraints as part of a table or column definition (however, see **Extensions**).
- REAL, DOUBLE PRECISION and NUMERIC data types
- WITH CHECK OPTION on a view definition
- WITH GRANT OPTION on a privilege definition

B.2 EXTENSIONS

An extension is a stand alone facility that is in X/OPEN SQL but not in "X3.135-1986" SQL. However, an implementation that supports the extensions will correctly process both conforming X/OPEN SQL programs and conforming "X3.135-1986" SQL programs.

X/OPEN includes the following extensions:

- The DROP TABLE statement
- The DROP VIEW statement
- The DROP INDEX statement
- The ALTER TABLE...ADD...statement
- The REVOKE statement
- The CREATE [UNIQUE] INDEX statement (as level 1 implementation of the UNIQUE constraint for a table)
- Data definition statements CREATE TABLE, CREATE VIEW and GRANT allowed in host programs
- The INCLUDE SQLCA declarative
- The definition of specific fields within the SQLCA.
- WHENEVER SQLWARNING
- Ownership by user-name (as level 1 implementation of explicit schema and associated authorisation-identifier)
- Embedded SQL in C
- Embedded COBOL host variables with data types corresponding to SQL SMALLINT, INTEGER and DECIMAL

B.3 DISCREPANCIES

A discrepancy is a deviation between X/OPEN SQL and "X3.135-1986" SQL with respect to a facility provided by both definitions. However, an implementation that supports an appropriate definition in each case, in general, the superset definition, will correctly process both conforming X/OPEN SQL programs and conforming "X3.135-1986" SQL programs.

- Character Data Type

"X3.135-1986" provides CHARACTER and the synonym CHAR and has optional length specification (default 1). X/OPEN only allows CHAR, and length is mandatory.

- Long Integer Data Type

"X3.135-1986" provides INTEGER and the synonym INT. X/OPEN only allows INTEGER.

- Decimal Data Type

"X3.135-1986" provides DECIMAL and the synonym DEC. "X3.135-1986" has optional precision and optional scale if precision is specified. X/OPEN only allows DECIMAL and has mandatory precision and scale.

- Floating Data Type

"X3.135-1986" allows optional precision with FLOAT (which must be at least equalled by the implementor). X/OPEN has fixed precision for FLOAT so a precision may not be specified.

- Indicator Variables

- i. "X3.135-1986" allows the keyword INDICATOR to optionally precede an indicator variable. X/OPEN does not support the keyword INDICATOR.
- ii. In "X3.135-1986", a host variable reference that is supplying a comparison value may contain an indicator variable to indicate the null value. X/OPEN does not support indicator variables in this context.

- Quantified Predicate

"X3.135-1986" supports SOME as a synonym for ANY. X/OPEN does not support SOME.

- GRANT

- i. "X3.135-1986" requires the keyword PRIVILEGES to follow ALL when specifying all privileges. X/OPEN does not support the keyword PRIVILEGES.
- ii. "X3.135-1986" allows a privilege definition to define a privilege already defined by the schema definition. X/OPEN does not allow a GRANT statement to grant a privilege already granted.

- DECLARE CURSOR

- i. "X3.135-1986" allows the keyword ALL to qualify the keyword UNION so as to provide for the retention of any duplicate rows resulting from the UNION operation. X/OPEN does not support ALL in this context.
- ii. "X3.135-1986" allows parts of the query expression to be parenthesized. X/OPEN does not support this use of parentheses. (Using parentheses in this way has no semantic implications.)
- iii. In order to execute an UPDATE CURRENT or DELETE CURRENT statement for a cursor, X/OPEN requires the cursor declaration to contain a FOR UPDATE clause. "X3.135-1986" neither requires nor allows the FOR UPDATE clause.
- iv. In "X3.135-1986", corresponding result columns of union-compatible query specifications must be named columns and have identical descriptions. In X/OPEN, corresponding result columns must additionally have named data types.

- Error Handling

"X3.135-1986" simply requires the existence of a host variable named SQLCODE. X/OPEN explicitly requires the use of the SQL communication area, SQLCA, which contains SQLCODE as a field.

- WHENEVER

"X3.135-1986" supports GO TO as a synonym for GOTO. X/OPEN does not support GO TO.

- Serializability

"X3.135-1986" guarantees that the execution of concurrent transactions is serializable. X/OPEN only guarantees an isolation level of cursor stability; therefore, repeatable reads are not guaranteed.

- Numeric Assignments

"X3.135-1986" does not allow an approximate numeric value to be assigned to a table column or host variable with a data type of exact numeric. X/OPEN does allow such an assignment.

- Embedded SQL Statements

In X/OPEN, an SQL statement embedded in COBOL must be the only construct within areas A and B of a line. "X3.135-1986" does not have this restriction and specifically allows a paragraph name to immediately precede an SQL statement embedded in COBOL.

- Comments

"X3.135-1986" allows comments in embedded SQL and defines the format to be used. X/OPEN does not allow comments in embedded SQL. However, comments may be inserted immediately before or after an embedded SQL

statement in the format appropriate to the host language.

- Name Lengths

In "X3.135-1986", an authorisation identifier has a maximum length of 18 characters. In X/OPEN, a user-name is equivalent to an authorisation identifier and has a maximum length of only 8 characters.

- Case Significance

In "X3.135-1986", SQL identifiers and keywords must be specified in upper case. In X/OPEN, SQL identifiers (that is, user-defined names) and keywords may contain upper case letters and lower case letters and are case insensitive.

- Unique Constraint

In "X3.135-1986", columns specified in a unique constraint must not be defined to allow null values. In X/OPEN, a unique index (which is functionally equivalent to a unique constraint) may be created including columns which are defined to allow null values.

- Set Functions

"X3.135-1986" allows the argument of a set-function-reference to be a correlated reference. X/OPEN does not allow a correlated reference in this context.

- Underscore

In "X3.135-1986" an underscore character in an SQL identifier must be followed by a letter or a digit. In X/OPEN SQL, identifiers (i.e., user-defined names) may contain consecutive underscore characters.

Future Directions

The X/OPEN Group wishes to extend its definition of portable SQL in the following respects:

- In exception handling, a distinction should be made between an error (administrative action or program change necessary) and an exception (time-dependent problem, e.g. deadlock, which can be overcome by retry; or value-dependent problem, e.g. violating a UNIQUE restriction, which needs user reaction). For this purpose, the "condition" in the WHENEVER statement should be extended to embrace the result class SQLEXCEPTION. If SQLEXCEPTION is not specified, SQLERROR handling will apply.
For the condition "no data found" and for those conditions raising SQLEXCEPTION, symbolic constants for the values of SQLCODE should be provided. As an example for "no data found":

In COBOL: 88 SQLNOTFOUND VALUE IS +100

In C: #define SQLNOTFOUND 100

- Additional data types for date and time as well as value expressions for the current date and the current time together with appropriate arithmetic should be provided.
- In a query-specification and sub-query, the forms "SELECT table-name.*..." and "SELECT correlation-name.*..." should be introduced.
- The definition of which data types are UNION compatible should be changed in the following respects:
 - All numeric types to be compatible, numeric values to be converted as needed.
 - Character strings of all lengths to be compatible.
 - Literals and expressions to be allowed and included in the conversions.
 - Columns with and without a NOT NULL clause to be compatible.
- As noted in Appendix B, the X/OPEN SQL definition differs from the "X3.135 - 1986" standard in several respects. In many cases this is due to the non-conformance of current implementations. The X/OPEN Group wishes to improve compatibility with the standard and considers the most important deviations that should be rectified by implementations to be as follows:

"X3.135 - 1986" level 2 exclusions that should be included

 1. Explicit use of ALL in set functions
 2. Escape characters in the LIKE predicate

3. Unique constraint in a table or column definition
4. WITH CHECK OPTION on a view definition
5. WITH GRANT OPTION on a privilege definition

Discrepancies that should be eliminated

1. CHARACTER not allowed as a synonym for CHAR; length not allowed to be omitted.
 2. INT not allowed as a synonym for INTEGER
 3. ALL not allowed to qualify UNION
 4. GO TO not allowed as a synonym for GOTO
- For the storing of long text, the additional data type VARCHAR should be introduced. Columns of this data type would not be allowed to be referenced in a search-condition.
 - In addition to the current (static) embedded SQL, facilities for the dynamic construction of executable SQL statements ("dynamic SQL") should be defined.
 - In addition to the current integrity constraints on a single table (CREATE UNIQUE INDEX), facilities for the definition of integrity constraints between the contents of several tables ("referential integrity") should be defined.
 - To select the level of isolation between concurrent transactions which is appropriate for the specific application, facilities to control locking should be defined for the embedded SQL program.
 - In addition to the current possibilities to join two or more tables, facilities should be provided for joins that also retain rows from a table for which no matching rows in the other table exist ("outer join").